END
DATE
FILMED
8

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DEVELOPMENT OF VISUAL INTERFACE
ENHANCEMENTS
FOR PLAYER INPUT TO THE JTLS WARGAME

by

Stephen L. Lower

March 1987

Thesis Advisor   Joseph S. Stewart II

AD-A181 989

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
Unclassified	

2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT
	Approved for public release;
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	distribution is unlimited.

4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION	6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION
Naval Postgraduate School	Code 74	Naval Postgraduate School

6c ADDRESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code)
Monterey, California 93943-5000	Monterey, California 93943-5000

8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

The Development of Visual Interface Enhancements for Player Input to the JTLS Wargame.

12 PERSONAL AUTHOR(S)
Stephen L. Lower

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ___ TO ___	87 March	89

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Simulation, Computer Graphics, Window Management.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis examines the design and development of a desktop prototype of a computer wargame. The prototype specifically deals with the ability to format the Joint Theater-Level Simulation's Model Interface Program (MIP) into the visual interface format of computer graphics known as window management. In this case, the Apple Macintosh microcomputer, a desktop computer, was used as the operating system for implementation of this prototype. The development of the prototype is examined with respect to the current version of the MIP. The prototype development is based on software design applications which include design models, correlation of programming languages to operating systems, and a breakdown of the design into a modular format. The thesis concludes with recommendations for changes which can enhance the use of the prototype from both a technical and managerial standpoint.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION		
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified		
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL	
Joseph S. Stewart II	(408) 646-2493	Code 55	

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete

The Development Of Visual Interface Enhancements
For Player Input To The JTLS Wargame

by

Stephen L. Lower
Captain, United States Air Force
B.A., Missouri Western State College, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY
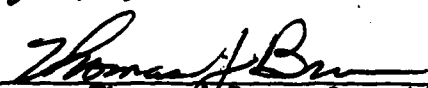(Command, Control and Communications)

from the

NAVAL POSTGRADUATE SCHOOL
March 1987

Author: _____
                          Stephen L. Lower
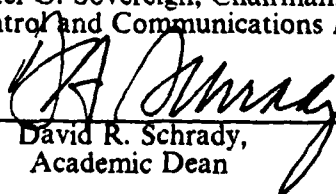
Approved by: _____
                    Joseph S. Stewart II, Thesis Advisor

_____
                    Thomas J. Brown, Second Reader

_____
                    Michael G. Sovereign, Chairman,
Joint Command, Control and Communications Academic Group

_____
                    David R. Schrady,
                    Academic Dean

2

ABSTRACT

This thesis examines the design and development of a desktop prototype of a computer wargame. The prototype specifically deals with the ability to format the Joint Theater-Level Simulation's Model Interface Program (MIP) into the visual interface format of computer graphics known as window management. In this case, the Apple Macintosh microcomputer, a desktop computer, was used as the operating system for implementation of this prototype. The development of the prototype is examined with respect to the current version of the MIP. The prototype development is based on software design applications which include design models, correlation of programming languages to operating systems, and a breakdown of the design into a modular format. The thesis concludes with recommendations for changes which can enhance the use of the prototype from both a technical and managerial standpoint.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

4

TABLE OF CONTENTS

6

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. PURPOSE OF THESIS

The purpose of this thesis is to examine enhancements of the human interface to an interactive computer simulation program by applying computer graphics techniques to the interface. These graphics techniques are known as the visual interface and have found widespread applications in man-machine interaction. In this study the viability of applying such an enhanced interface to an existing application is based upon three factors: 1) The casual user of such an application does not have or maintain the necessary skills to efficiently utilize the application. 2) The technology which supports the enhanced interface has advanced past the technology of the current application's design since the design was frozen and implementation of the design into a finished product was begun. 3) The low cost of computer's with large amounts of memory and extremely capable CPUs has led to a proliferation of advanced microcomputers. Given these factors, the development of an enhanced interface is achieveable.

The achievement of the enhanced interface requires a knowledge of background information about the interface subject. The subject is the Joint Theater-Level Simulation (JTLS) and it's user interface. By beginning with the purpose of JTLS and how it is utilized, the scope and organization of the research of this thesis may be established. The background information about JTLS and the scope of the research follow later in this section. Chapter 2 examines the current JTLS interface (the Model Interface Program), it's design and structure, it's current functions, and it's *modus operandi*. Chapter 3 discusses the design of an enhanced interface, the correlation of the current interface with the enhanced version, the enhanced versions *modus operandi*. concludes with proposals and observations about various aspects of the enhanced interface development and utilization. Chapter 4 concludes the thesis with a summarization of the enhanced interface prototype design and it's usage. These sections flow from the basic design and operation to an enhanced design and purported operation which is achieveable. The research begins with the background review of JTLS.

B. BACKGROUND

The goals and scope of this thesis are predicated upon understanding the object examined. The nature of the object, it's reason for being to include a brief review of it's history, and it's present technical configuration provide a basic foundation upon which this research may be built. The object is JTLS and it's nature is that it is an interactive computer simulation model used for wargaming. The original objectives behind the design of JTLS and how those objectives have evolved are discussed to provide a link to future JTLS use. Finally, the present technical configuration of JTLS presents it's hardware and software elements which are the backbone of JTLS implementation. A more detailed discussion of these elements follow.

Computer simulation is an effective, economical method of analyzing military plans and operations without actually employing the military forces which carry out those plans and operations. It is possible, using computer-based simulation, to trace, in detail, the consequences and implications of a proposed course of action [Ref. 1: p. 1-2]. One such simulation (or wargame) is the Joint Theater Level Simulation (JTLS).

A complete set of manuals documenting JTLS, from it's inception to it's implementation, has been published. Much of the following background information is taken from the JTLS Executive Overview manual. JTLS is a computer-assisted wargaming system that models two-sided air, ground, and naval combat. It was designed for use in warfare training, joint operational planning, and doctrinal analysis with an emphasis on rapid production of results, theater-independence, and ease of use for non-programmers. The original design objectives of JTLS were to 1) provide a contingency planning analysis tool for the United States Readiness Command, 2) provide an educational wargame and analytic capability for the United States Army War College, and 3) provide an analytic tool aiding contingency plan evaluation for the United States Army Concepts Analysis Agency.

In 1985, the Joint Analysis Directorate of the Organization of the Joint Chiefs of Staff assumed responsibility for the control and direction of future JTLS development efforts. (The Joint Analysis Directorate is now the Force Structure, Resource, and Assessment Directorate (OJCS/J8). This was done as part of a program to upgrade the analytic tools available to the unified Commanders in Chief for use in war planning. Included in their interests are future developments of JTLS which enhance user friendliness through advanced technologies such as the visual interface of window management.

The need for enhancements to the human interface are due to the nature of the use of JTLS. As an analytic tool it is used sporadically to test doctrine, strategies, and tactics by a variety of users. Frequently, these users are not computer operators by trade nor do they spend many hours playing JTLS. Their primary interest in JTLS is the outcome based upon a preformatted and staged input to the game. They do not need a complicated, difficult to learn (and easy to forget) computer simulation that is not readily usable when they are trying to develop and conduct an experiment with warplans. The Model Interface Program (MIP) can be such a program for the casual user. Unless the user frequently plays JTLS, the operation of the MIP is moderately difficult on which to maintain proficiency and it is not conducive to quickly resurrecting lost proficiency.

The JTLS system is designed to operate on Digital Equipment Corporation's VAX minicomputer systems, including the 11/780, 11/785, and 8600 computers. The minimum hardware configuration for JTLS consists of four video terminals and one on-line printer. The maximum configuration consists of 28 video terminals, 10 graphics displays, and one or more line printers.

The following support software is required to implement JTLS:

1) A VAX/VMS operating system.
2) A SIMSCRIPT II.5 programming language compiler.
3) A "C" programming language compiler.
4) An INGRES data base system.

Most of JTLS is written in the computer language SIMSCRIPT II.5 (a registered trademark of CACI, Inc.). The language is designed to facilitate the simulation of large, complex systems. The simulation constructs are embedded in the language, which is free-form and English-like. SIMSCRIPT II.5® also has such automated features as statistics-gathering mechanisms, dynamic storage management, and flexible report generating. For these reasons and others, SIMSCRIPT II.5® was selected as the high level programming language for the simulation applications. [Ref. 2]

JTLS may be summarized as being a complex, sophisticated set of computer programs which may have more extensive capabilities when properly configured and used. With these facts about JTLS in mind, the examination of the player interface to the JTLS may be conducted.

C. SCOPE

The Joint Theater-Level Simulation (JTLS) is an interactive computer simulation model used for wargaming of theater conflicts. The nature of a two-sided computer simulation, such as JTLS, is to produce an outcome as the result of the interaction between the two opposing sides. In this case, the two sides simulate combat by directing simulated military forces into proximity, with the result being an outcome of a battle. The whole impetus behind this simulation is the involvement of the player, i.e., the human interface. The main area of interest in this study is the Model Interface Program (MIP). It is through this program that the human interaction with the game is conducted and is what the prototype design will enhance.

Several avenues of research may be taken to develop the aforementioned prototype. The method used here is to develop the prototype using as much of the existing MIP source code as possible. This approach provides an economical, quick ability to implement a full-scale visual interface. The efficiency of such a prototype compared to a total redesign of the MIP in terms of future expansions or computer use will not be addressed in depth.

The general operation of a JTLS simulation is to conduct an interaction within the combat simulation by issuing orders to the available military forces. The Combat Events Program (CEP) compares the actions of the forces, within the limitations of their environments, and yields the results of the combat. As a result of the interaction, the commanders of the forces must continually make decisions during the course of the game. These decisions (the issuance of orders) are transmitted to the CEP via the Model Interface Program (MIP). Thus, the MIP is an interactive program used by all players.

The present version of the MIP, while fully capable of interacting with the CEP, is considered unwieldy for the occasional user, difficult to learn, and slow in terms of conducting a fast paced simulation. Of primary concern is the methodology of issuing orders to the CEP. While this methodology is examined in depth later in this thesis, it may be safely stated that the MIP lacks user friendliness for the occasional user and is not meeting current and future requirements for the purposes of player interaction with the game.

One method of enhancing player interaction to JTLS is the use of the visual interface. The visual interface was borne out of the "need for creating easy-to-learn and easy-to-use applications" [Ref. 3]. Some advantages of the visual interface are to

12

increase the data absorption rate by the user, reduce input/output errors such as those which occur during the typing of data, provide the user a "positive transfer of learning" to the new system, and the reduction of user anxiety caused by a lack of control or a lack of information [Refs. 4,5]. The visual interface simply allows the user to "see" what is going on; a much faster process than "reading" what is going on.

Implementing the visual interface is done through a variety of computer graphics applications. The visual interface application examined in this thesis is that of window management. Window management deals specifically with the methods of creating graphical forms (windows), and displaying/manipulating information within those windows. The Apple Macintosh[tm] is an excellent machine for implementing the use of window management techniques in microcomputer application programs.

The scope of this thesis will be to investigate the current methodology of creating a player order (a directive) and issuing it to the Combat Events Program(CEP), breaking down the methodology to create and issue the order, and reconstructing the methodology using the visual interface format. The particular case study will create an air directive, with it's associated utility directives, and send it to the game. In doing so, much of the material to follow will examine particular commands and directives as representative functions of the overall MIP. The methodology developed and used in the course of designing the prototype will be a useful tool in the expansion of the prototype to include all MIP functions in a Macintosh interface. In the opinion of the author, the basic constructs of the visual interface prototype should also be useful in the event of a total redesign of the MIP. The study begins with a detailed examination of the current Model Interface Program.

II. THE MODEL INTERFACE PROGRAM

A. THE RELATIONSHIP BETWEEN JTLS PROGRAMS

The JTLS Executive Overview addresses the overall structure of JTLS. The JTLS system consist of several independent programs which work together as a system of functions. The functional programs of JTLS are described below. The functional programs of JTLS have a variety of interrelationships as shown in Figure 2.1. It is through these relationships that the MIP initializes itself; obtains data from databases, files, records, and displays; and performs its functions.

The functions of the Model Interface Program are:

1) Entering orders.
2) Processing orders.
3) Communication between players and game controllers.
4) Communication between the players and the combat simulation.
5) Accessing and using support information.
6) Saving directives in archive files.
7) Analyzing post-processor data.
8) Controlling graphics output.
9) Stopping or temporarily halting the game.

To accomplish any of these functions the MIP must depend on the other JTLS programs for support. For example, the CEP and Executive Program provide the information required by the MIP to create and process orders. The game's scenario database, which provides the players units, equipment, etc., is created by the Scenario Preparation Program. While the MIP does not communicate directly to all JTLS programs, it does have an indirect relationship to those outlying programs. In concept, since the MIP is a critical function of the execution phase of JTLS, its importance demands the support of the other programs and, in turn, results in the relationships shown. [Ref. 2]

B. THE MIP STRUCTURE

The structure of the MIP can be derived from its functions, its relationships, and the high level language SIMSCRIPT II.5® used to program the MIP. In deriving the structure, several system models were developed to express the why, what and how of the MIP. These models are discussed below.

14
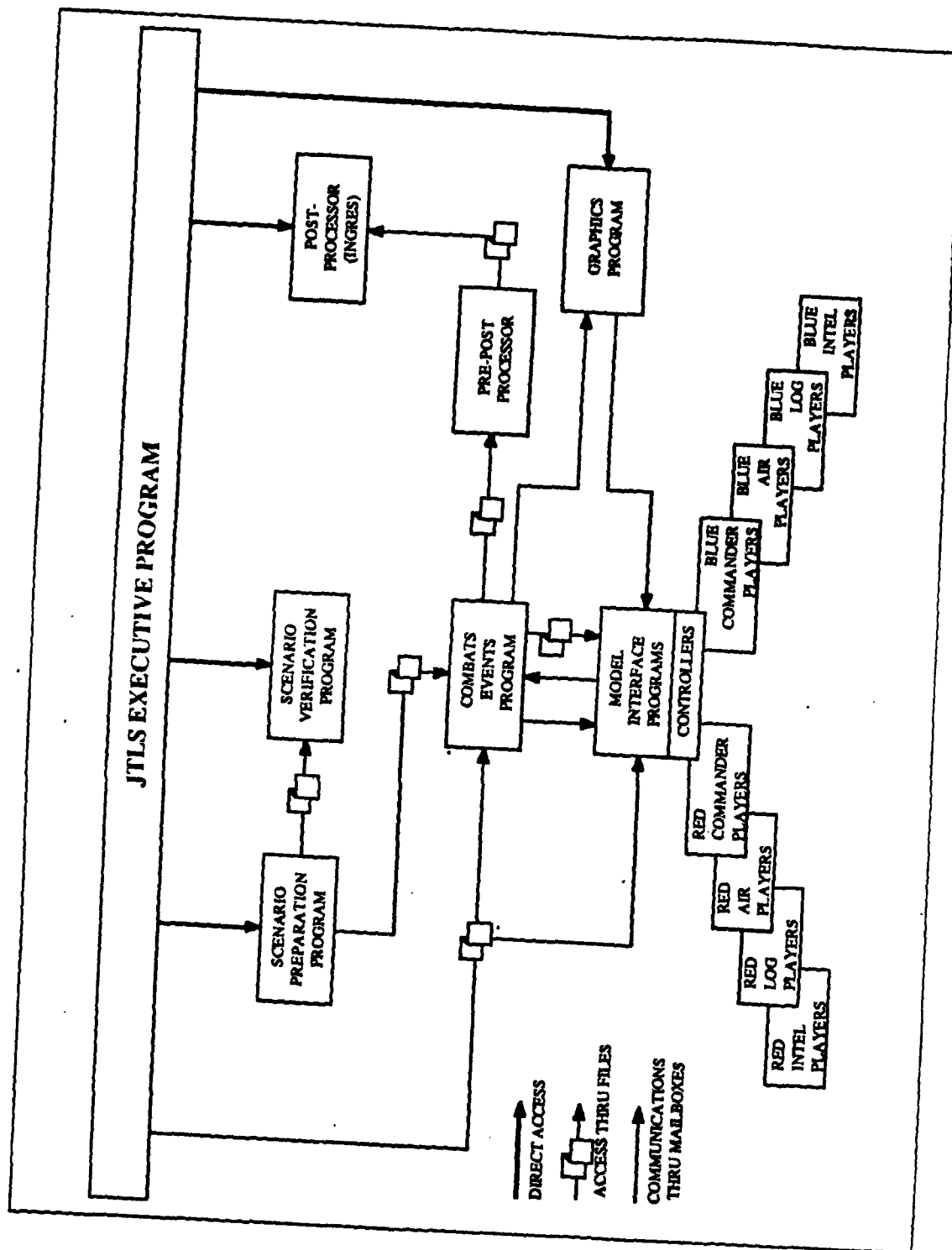
Figure 2.1   Functional Programs of JTLS.

15

1. The Fundamental Model

The overall system model is the fundamental model, Figure 2.2. In this model the various user inputs to the program are shown as well as the outputs of the program. Note that the functions are not delineated here since they are inherent to the MIP in this model form. The purpose of the fundamental model is to define the desired results and identify the necessary inputs while leaving the identification of specific contributors to any given function to the program. The "how" is examined in greater detail through data flow diagrams.
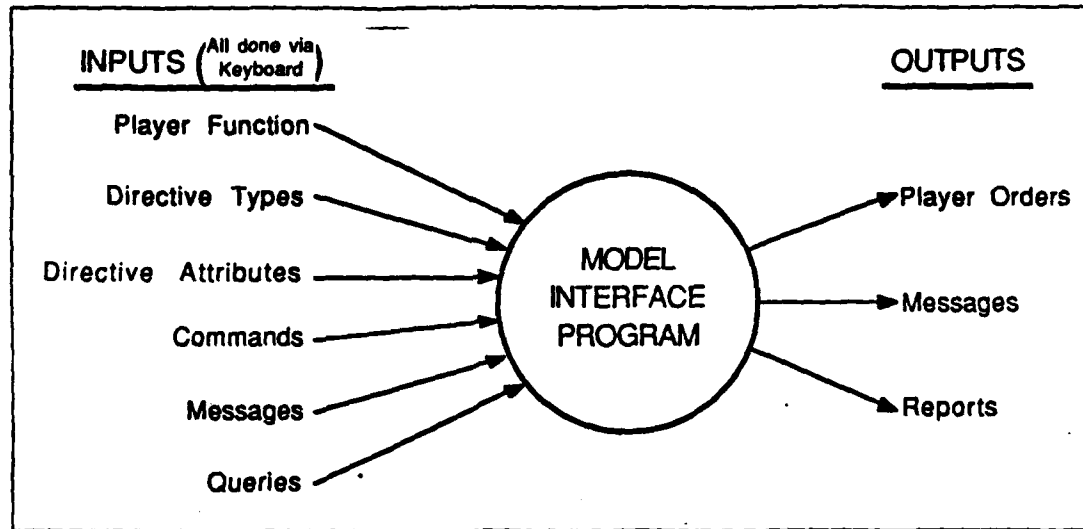
Figure 2.2   The Fundamental Model.

2. The Data Flow Diagram

The data flow diagram, Figure 2.3, displays the flow of information from the player to the game. Although not intricately detailed here, it does show the basic transformations which take place, what type of information is passed, and the location of sources or sinks of information.

A significant portion of the flow of information from the player (the input) to the game (the output) deals with the directive. The directive is the heart of the game in that it literally causes an interaction to take place in the game thus producing some outcome. Without the directive, there would be no simulation model. Due to the directive's importance to this application, much of the design is described with the directive as it's focal point. To appreciate the directive's impact on the flow of

16
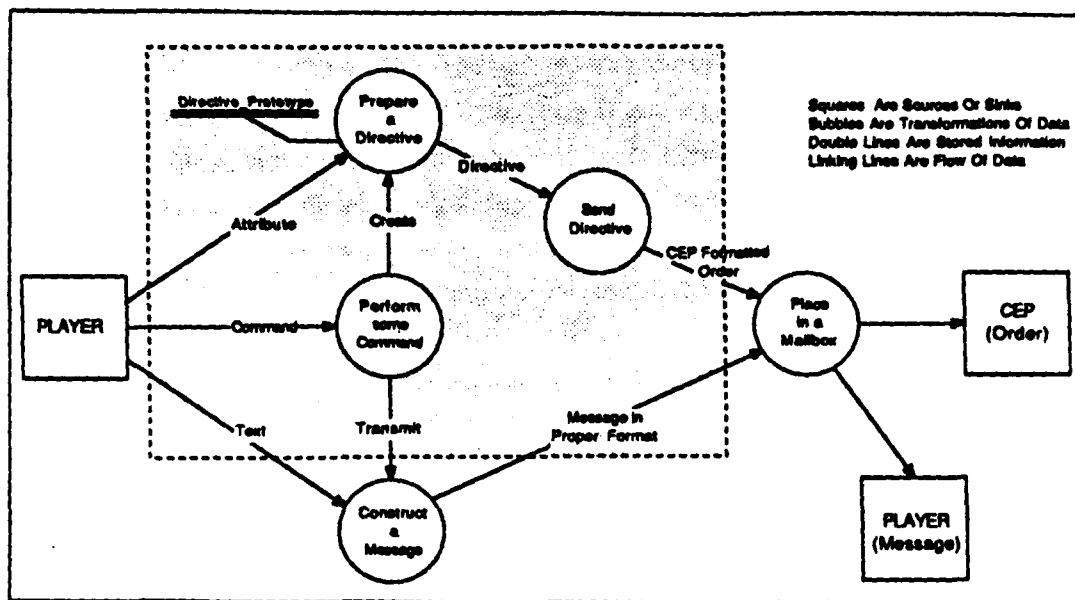
Figure 2.3   The Data Flow Diagram of the MIP.

information, one must understand that most of the data is manipulated with the goal of developing and exercising a directive.

At this point it is necessary to understand the performance of the various elements of the data flow diagram. In the case of the player creating an order, the player first enters a command. The transformation of the input is the performance of that command. If the command was a *create* command then the next transformation would build the directive using the attributes entered by the player. When all the attributes have been entered, the player enters another command to tell the MIP the directive is completed and to manipulate the directive. For example, this could be a *verify, hold, save,* or *send* command. Any command other than a *send* command would require the player to enter more commands before the directive could be sent to the CEP. When the player issues a *send* command, the directive is formatted into an order message the CEP can read and understand. The order message is then placed in the CEP's mailbox.

A key issue here is that the player is constantly entering data directly into transformation modules without the data flowing in a "fluid" manner towards an output. The superimposed box outlines a bottleneck of data flow in the data flow diagram. This bottleneck places a great demand on the player to provide data in this

17

model. The source of the player's data is a computer printout of function specific information printed prior to playing the game. This is undesirable since all (or nearly all) of the data necessary for transformation during the creation of a directive is available in a database or file in the game itself. In particular, a file called the Player Initialization File (PIF) exists for each player and contains much of the information needed by that player to perform transformations, i.e., developing directives. Ideally, the transformation mechanism , vice the player, would do the work of sourcing and entering the data. Such a mechanism can be created using the graphics interface environment.

3. The Data Structure

The data structure of the MIP is a hierarchical system that is implemented by using multi-linked lists containing scalar items, vectors, and n-dimensional spaces. In SIMSCRIPT these concepts are established first as *entities*. These entities are characterized by their *attributes*. If there are logical associations or groupings of entities, they are described as *sets*. [Ref. 1: p. 1-15]

A set is a logically ordered collection of entities organized through a system of set pointers. A pointer is the address in memory of a data item. For example, the MIP has a set of targets with pointers to (i.e. the addresses of) the first and last members of the set and the number of members (targets) in the set, Figure 2.4. These attributes of the set are considered to be *owner* attributes. Each member (target) has pointers to (addresses of) the preceding and succeeding members of the set as well as a flag to record membership in a set (to disallow multiple membership).

Entities may be permanent or temporary. The permanent entities exist throughout the simulation unless they are explicitly destroyed. Temporary entities are those which are short-lived during the simulation or for which the number of copies varies within the execution of the simulation.

Figure 2.5 shows an example of some permanent and temporary entities. While AIRCRAFT_(1) and AIRCRAFT_(2) exist in storage throughout the game, the RECOGNIZED_COMMAND is only put into storage at the time it is created.

All of the data *used* in the game by the MIP is stored in these sets, entities, or attributes. Their definitions may be found in the MIP's source code preamble. All of the data *needed* to create player directives is found in these data structures (primarily in the PIF). However, the data structures aren't effectively used. This was demonstrated in an earlier section and will be discussed later in this thesis. For example, the

18
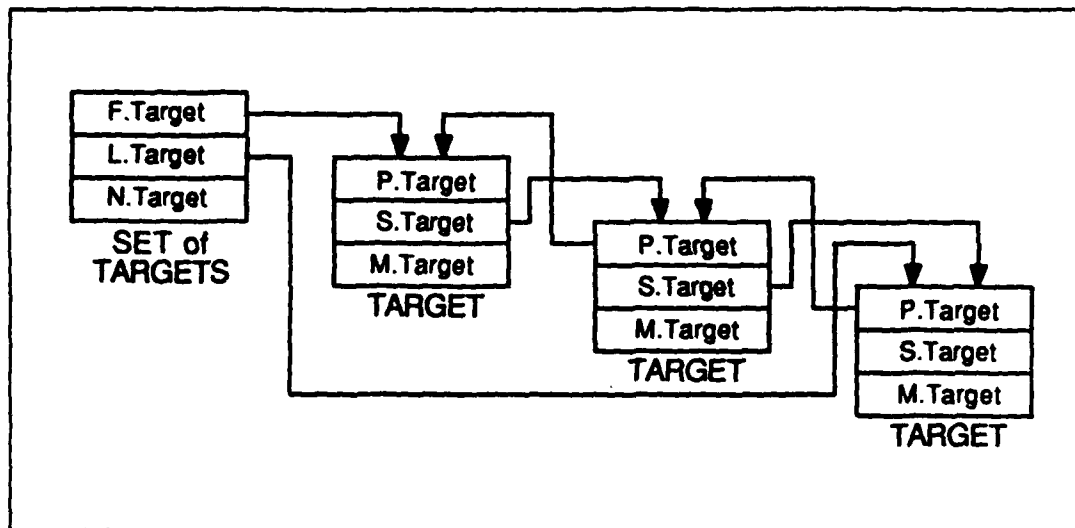
Figure 2.4   An Example of Set Organization.
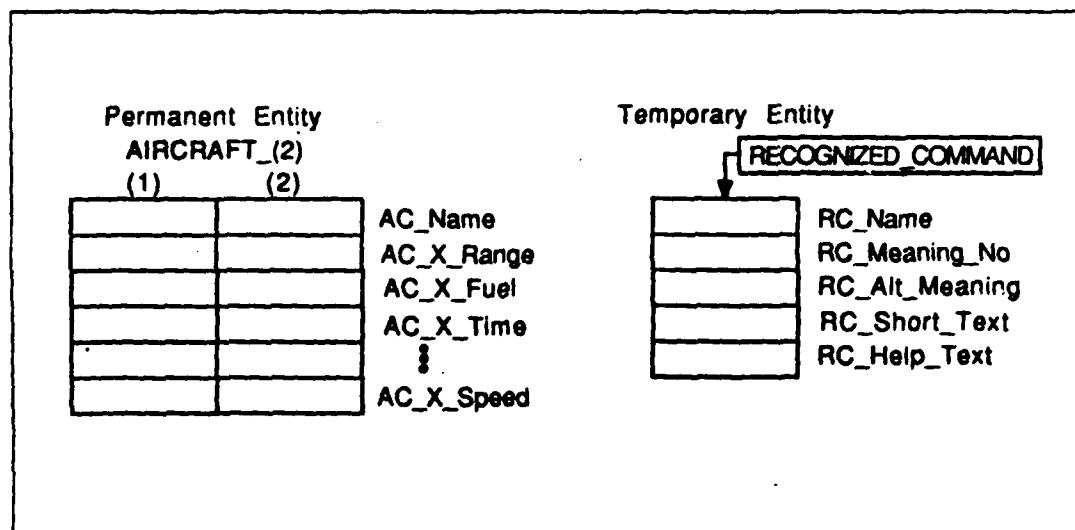
Figure 2.5   Examples of Entities.

AWACS air directive only uses data from the permanent and temporary entities listed in Figure 2.6. The PIF's function here is simply error checking to ensure the data entered is correct with respect to the scenario's data. The poor use of the PIF results in the increased burden of furnishing data being placed upon the user.

19

```
        Permanent  Entities      Temporary  Entities
        Directive_Prototype         Directive_
        Attribute_Prototype
        Create_Routine
        Word_Indicator
        ───────────────────────────────────────────

        NOTE:  These do not include type
        checking,  help,  or  graphics  entities.
```
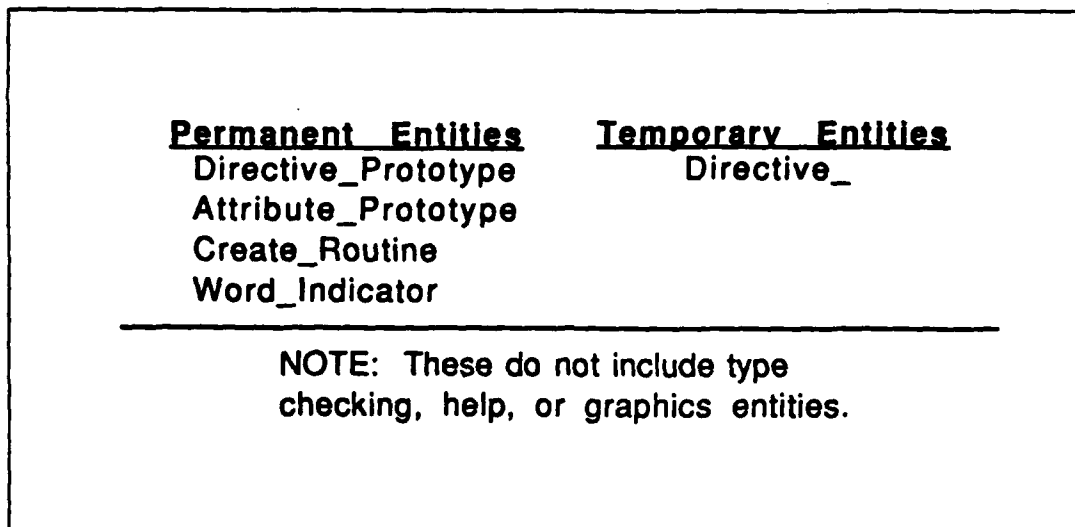
Figure 2.6 The AWACS Directive Data Structure.

C. THE MIP FUNCTIONS EXAMINED IN THIS PROTOTYPE

1. The Commands

The MIP has 36 commands which perform environmental, directive specific, or group of directives specific tasks for the player. These commands essentially instruct the MIP to take further actions which have been defined by the player to accomplish his decisions. The environmental commands perform the administrative tasks such as printing, spooling, scanning, finding, etc. The other two categories of commands manipulate the directive(s) by creating, changing, sending, etc. At times, the delineation of these commands into the three categories seems vague. However, this delineation will become clear later when the visual interface is applied to the MIP functions.

The specific commands of interest and their definitions are as follows:

- **CREATE**   The *create* command allows the player to create a directive within the domain of the player's function.

- **GCREATE**   The *gcreate* command allows the player to create a group of directives within the domain of the player's function.

- **QUERY**   The *query* command allows the player to request that the CEP send the player standardized reports which pertain to the player's function.

- **FIND**   The *find* command allows the player to hold a directive that was previously in existence.

- **COPY**   The *copy* command allows the player to create a directive whose attributes, except for the identifier, are identical to those of an existing directive of the same type.

20

- GCOPY   The *gcopy* command allows the player to create a group whose attributes, except for the identifier, are identical to those of an existing group.

- JOIN   The *join* command allows the player to place a directive into a group.

- LEAVE   The *leave* command allows the player to remove a directive from a group.

- GCLEAR   The *gclear* command allows the player to empty a group of all it's directives.

- DISPLAY   The *display* command allows the player to see which directives of a particular type have been created in the past and still exist, i.e., have not been deleted.

- GDISPLAY   The *gdisplay* command allows the player to see which directives are in a particular group

- MENU   The *menu* command allows the player to view the menu of any existing directive without holding onto it.

- SAVE   The *save* command records all of the players undeleted directives onto a file called the Archive File.

- LOAD   The *load* command is used to bring directives from an Archive File into the MIP.

- TRANSMIT   The *transmit* command is used to transmit messages to other players.

- SCAN   The *scan* command allows the player to view several messages in succession.

- SPOOL   The *spool* command allows the player to put several messages in his or her print file without taking the time to examine them.

- PRINT   The *print* command prints out a hard copy of all messages filed or spooled.

- REFRESH   The *refresh* command brings up a fresh MIP screen.

- SET   The *set* command allows the player to set various parameters that tailor the use of the MIP to the player's liking.

- ADJUST   The *adjust* command allows the player to adjust the display of his graphics station.

- RETURN   The *return* command is used in conjuction with the exclamation mark to allow the player to use the *OVERMIP* feature. The return command conveys the player's intent to abandon the current overmip and resume action that the player had previously interrupted with the most recent exclamation mark. (NOTE: The *OVERMIP* feature freezes the current player action allowing the player to perform some other action and then return to the point where the frozen action was interrupted. Not all player functions can be performed in the *OVERMIP* feature.)

- SEND   The *send* command is used to send the information contained in a directive to the CEP in the form of a *player order*. The command only applies to the held directive, if there is one. (NOTE: The *send* command applies only to action directives. Utility directives cannot be sent to the CEP explicitly; rather, the information contained in them is sent as part of the action directives that refer to them.)

21

- **GSEND**   The *gsend* command is used to send, one at a time, all of the directives belonging to a particular group.

- **CHANGE**   The *change* command is used to change specific attributes of the held directive after its initial creation.

- **RESTORE**   The *restore* command allows the player to override all change commands performed in the held directive since it was last held.

- **PAGE**   The *page* command lists the menu of the held directive and, if the directive menu is contained on more than one page, it will cause the MIP to enter the paging mode.

- **VERIFY**   The *verify* command performs all validation checks not performed during the creation of a directive or the changing of attributes.

- **GVERIFY**   The *gverify* command performs the verify command for each directive in a particular group.

- **DELETE**   The *delete* command permanently removes the held directive from the MIP.

- **GDELETE**   The *gdelete* command permanently eliminates a group of directives from the MIP.

- **DONE**   The *done* command returns the MIP to a state in which it is not holding any directives.

2. The Directives

The directives are essentially the actions the player wants to take in the course of playing JTLS. They tell the game what unit will take what action at what date/time with what resources. When the player begins to *create* a directive, a **template** appears on the terminal screen listing the **attributes** that comprise the directive. The directive template displays indicate if a data input for a particular attribute is optional or mandatory.

While all directives contain attributes, those attributes only consist of a few basic types. The data input to those attributes are the distinguishing factors among directives. The most frequently encountered attributes are as follows:

- **REFERENCE**   A player selected name which *uniquely* identifies the directive.

- **UNIT, SQUADRON**   The name of the unit or air squadron being given the directive.

- **TIME**   The time for the directive to be implemented by the CEP.

- **DURATIONS**   The number of days, hours, and/or minutes the directive action is to be conducted.

- **COORDINATES**   The latitudes/longitude pairs which indicate geographic points of interests (for a variety of reasons) in the directive.

- **ROUTE, LOAD, LIST**   These are utility directives used as attributes in action directives. They must be created before an action directive may be sent to the CEP and implemented.

22

One extremely useful feature of JTLS is the ability of the graphics system to send names of units and targets and latitude/longitude points to the MIP. When graphics is used to enter any of these attributes, the MIP acts as if the player entered that data. A shortcoming of this feature is that the player has to establish a communications link between the MIP and the graphics station used. This must be done at *both* the player and graphics terminals.

a. Creating the Action Directive

To fully understand how the creation of a directive is accomplished, the reader should step through the process of directive creation. For example, the AIR player would enter the *create* command. If the player didn't know the type of directives he or she could create or didn't know the proper syntax for the name of a directive, the player could enter a question mark (?). The MIP would then display a list of the air directives and associated utility directives. From that list the player would determine the type of directive to create, enter a "Q" to quit viewing the list, and when prompted, enter the name of the directive.

Upon receiving the directive type, for example AWACS, the MIP would display the AWACS directive template on the terminal screen, Figure 2.7 . Of the nine AWACS attributes, three of them are utility directives. Five of the attributes have their data values checked for validity when the *verify* or *send* commands are entered to the MIP. As the player begins to enter data, each attribute is sequentially entered as a single entry or, for the expert player, as a stack of entries. At this point, close examination of the AWACS directive creation will show the reader what the MIP is doing during the process.

The first attribute is the **Mission.** This must be a unique identifier to distinguish this directive from other air missions sent to the CEP. The MIP help function (the ?) describes the format for the identifier. When verifying or sending the directive, the MIP will check the identifier for uniqueness.

The next attribute is the **Squadron.** This must be the name of a squadron type unit that the air player has under his or her auspices. Only a syntax check is performed here.

Aircraft is the third attribute. This is a number that cannot exceed the number of aircraft in the squadron. The MIP will accept any value during data entry, but will match the value to the **Squadron** when the *verify* or *send* commands are entered.

23

```
MSG: 0          0.000  TO  1         040000ZJUL85    0.0000        NIGHT

AWACS (AW) DIRECTIVE:

   1. MISSION:            xxxxxxxx          8. ORBIT LAT/LON:     dd-mm-ssD  ddd-mm-ssD

   2. SQUADRON:           xxxxxxxxx         9. SENSOR LIST:       xxxxxxxx
   3. # AIRCRAFT:         nnnn

   4. ROUTE IN:           (xxxxxxxx)
   5. ROUTE OUT:          (xxxxxxxx)

   6. ORBIT ENTRY TIME:   ddhhmmZMMMYY
   7. ORBIT DURATION:     ddDhhHmmM




   MIP COMMAND: CR  AW
   MISSION:
```

Figure 2.7   The AWACS Directive Menu.

Route In and Route Out are attributes which are utility directives. The data values of these attributes are the names (identifiers) of routes created separately using the Air Route directive. These are checked to determine if the routes exist when the *verify* or *send* commands are entered.

The Orbit Entry Time attribute is a time for the AWACS mission to begin surveillance in its flight pattern. It is entered as a date-time-group sometime in the future. When the game receives the directive it takes into consideration the time it takes for the aircraft to reach the orbit pattern and the time it takes to prepare the aircraft for launch when determining the validity of this time. If the squadron doesn't have enough time to prepare, launch, and fly the aircraft to the orbit pattern by the assigned time, the game will advise the player of that fact. The only real-time check is for syntax.

The Orbit Duration attribute is a time which tells the game how long the aircraft will orbit in its pattern. This time is checked by the game by comparing it to the crew's maximum allowable flight time and advising the player if the duration is too long. The only real-time check is for syntax.

24

The **Orbit Pattern** is entered as a set of latitude/longitude coordinate pairs. The coordinate pairs determine the two end points of an elliptical orbit pattern. The only real-time checks are to determine if the points are on the surface of play and for syntax.

The **Sensor List** is a utility directive. The data value of this attribute is the name (identifier) of a list of sensors to be loaded onto and used by the aircraft. The sensor list directive is created separately. The attribute is checked to determine if the list exists when the *verify* or *send* commands are entered. Since this is also the last attribute, the player must enter some command to manipulate the directive. It must be noted that this is the "held" directive until the directive is manipulated in some manner which "unholds" it.

b. Creating the Utility Directive.

Utility directives, as previously mentioned, are created separately from action directives. They must exist when the player attempts to *verify* or *send* to the CEP a directive which references them. There are two avenues to create a utility directive. One is to create the directive when the player has a blank screen. The other is to use the *OVERMIP* feature while creating an action directive, suspending the player's interaction with the action directive, and allowing the player to then create the utility directive as if a blank screen existed.

The **Air Route** directive has two apparent attributes as shown in Figure 2.8. One is the *Route ID* which is unique to that route. The other is the *Latitude and Longitude*. This coordinate pair is entered for every point of the air route except the origin. A null entry (NE) is entered after the last pair. The MIP then prompts the player for altitudes for each point. Altitudes are from 500 to 60,000 feet. A null entry is then used to quit.

The **Sensor List** directive, Figure 2.9, specifies the sensors to be included in a particular sensor package configuration used for various air directives. The two attributes of this directive are the *List ID* and *Sensor*. The List ID is the unique identifier of that list. The sensor is a category of sensors which indicate which type of sensors to put into the list. A null entry is used to quit.

3. Summary

This section has described the relationships between the programs which constitute JTLS, the structures of the Model Interface Program, and the MIP functions to be examined in the prototype. One very important aspect of JTLS which will have

```
MSG: 0         0.000  TO  1         040000ZJUL85    0.0000        NIGHT
                                                                           
AWACS (AW) DIRECTIVE:

  1. ROUTE ID:            xxxxxxxx

    2. LATITUDE          LONGITUDE              ALTITUDE




  _____



  MIP COMMAND: CR ARTE
  ROUTE ID:
```

Figure 2.8   The Air Route Directive.

```
MSG: 0         0.000  TO  1         040000ZJUL85    0.0000        NIGHT
                                                                           
SENSOR LIST (SL) DIRECTIVE:

  1. MISSION:            xxxxxxxx            2. SENSOR




  _____



  MIP COMMAND: CR SL
  LIST ID:
```

Figure 2.9   The Sensor List Directive.

an impact on the approach to development of this prototype has not been addressed. That is the operating system and sequence of execution in SIMSCRIPT II.5® The foundation of the SIMSCRIPT system and the sequence in which JTLS (and the MIP) source code is executed is the basic timing routine inherent to SIMSCRIPT, Figure 2.10.

Execution of a SIMSCRIPT program begins with the first statement in the *MAIN* program and continues through a series of steps. Resources must be created and initialized before they are used by *processes.* Then the initial processes are activated in *MAIN* (since SIMSCRIPT requires that something be awaiting execution before a simulation commences). A *simulation* begins when control passes to a system-supplied timing routine. This is done by executing the *START SIMULATION* statement. The significance of "something must be awaiting execution" is understood when the main program is examined.

The *MAIN* Program contains several processes. One of these is the *terminal* process. This process is literally the keyboard read process, i.e., how the player inputs data through the keyboard to the MIP. When the player uses the keyboard, the process is activated. In terms of the timing routine, this means that the process is placed on the pending list and is executed by the timing routine. When the player's keyboard is idle (the player has used the return key to enter something), the process is not on the pending list and the timing routine waits for another process to be placed on the pending list. During this idle time, the MIP (and operating system) are essentially waiting for the player to do something in the interactive mode. Here the MIP can still be performing some non-interactive tasks. The significance of this idle time created by the MIP is a temptation to the designer to interface directly with the MIP rather than the system. It will be demonstrated in Section III that this is not a particularly effective approach for development of this prototype.

D. A COMPARISON OF SOURCE CODE VERSIONS

To further illustrate the operational behavior of the Model Interface Program, a comparison was made between the current version of the MIP source code and the source code of a prototype version. In the comparison, a particular objective was selected to be accomplished by the source code. Both versions of source code began at the same point and finished with the same result. The current version is written in SIMSCRIPT while the prototype version is written in Pascal for operation on the

27
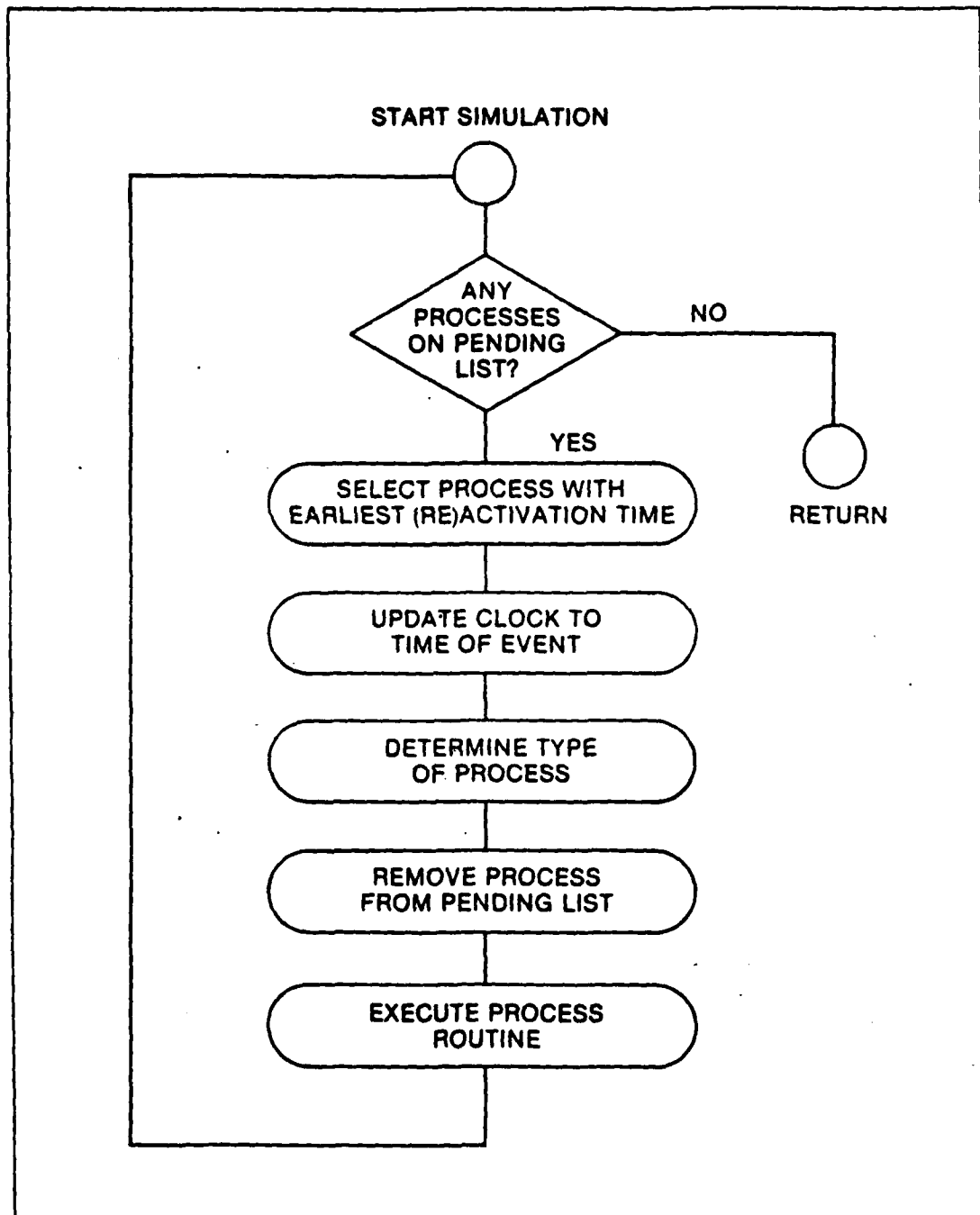
Figure 2.10   The Basic SIMSCRIPT Timing Routine.

Macintosh. Tables 1, 3, 5, and 7 are the current versions of source code for steps 1-4 respectively. Tables 2, 4, 6, and 8 are the respective prototype versions of source code. There are several noticeable differences between the two sets of source code. These differences will be pointed out in the following narrative.

The comparison was made using the creation of a Sensor List directive as the objective of the source code. Both versions of source code begin that process at the point where the player must select the directive type. The process then is broken down into a series of steps. Step 1 is to select the type of directive to be created. In this case, Sensor List is selected. Step 2 is to display the directive on the screen. Step 3 is to assign an identification reference to the directive. Step 4 is to assign sensor packages to the sensor list. The process ends at this point. From here, for example, the player could save, verify, or send the completed directive.

It should be noted that in the current version the steps must be taken in strict sequence. The prototype version permits the reversing of sequence order for steps 3 and 4. The order of sequence is left strictly to the player's discretion as to what step to do when. The player can even go back and redo a step in the prototype version. This is not permitted in the current version. To do so the player must exit this process after it is completed and begin a totally different process.

A significant assumption was made regarding this process. This should be noted so the reader may gain a greater appreciation for the results of the comparison. First, it is assumed that the player will always enter syntactically correct, accurate data. Thus, format and type checking source code has been left out of the example in the current version. The prototype incorporates the checking into its code due to the nature of its operating system. Except for one case, no prototype case requires any explicit checking.

It was also assumed that the player would not abort the process. The current version source code to do this was also left out. The prototype version did not require explicit code as this is inherent to the operation of the system. Also, any code dealing with error messages to the player was deleted from the listings. The current version has quite a few error messages while the prototype version would only require one for this process and its message is inherent in the operating system. The "help command" code was also omitted from the current version. It literally is not needed in the prototype version.

29

A readily evident result of the comparison now exists and should be mentioned despite the risk of prejudicing the overall outcome. The result is that quite a reduction of source code can apparently be made between the two versions in the areas of checking, process abort, and error messages. This is not a hard and fast result in the final outcome however. The reduction in source code now may be offset by an increase in code to perform other functions later. It is the opinion of this author that this will not be the case.

1. Step 1

The process of selecting the directive type is straight forward. The presentation of information to the player requesting a selection is quite different. The current version presents a blank screen in the content region (the middle of the screen) while the scroll area (the bottom portion of the screen) contains the prompt "directive type:" with a blinking cursor a few spaces to the prompt's right.. Here the player would begin the process by typing in the words "sensor list". The prototype version presents the player with a box in the middle of the screen. The box contains a list of directive types which are currently available to the player. The player moves the cursor to the "sensor list" item in the list and selects it.

The determination of what type of directive to create has been completed. The type selected in both versions was the Sensor List. A breakdown of the step reveals several interesting contrasts between the versions. The first is the display itself. The current version puts it's information for the player near the bottom of the screen. While it is out of the way for the main portion of the screen, it is also "out of the way" in terms of the player's visual focal point. In general, a person's initial focal point is the middle of a display and then the person examines the display area to seek out the required information.

The prototype version places it's display in the middle of the screen which is the player's initial focal point. The player doesn't have to search for the information. This contrast is subtle, but nonetheless significant throughout the process and in terms of information transfer to the player.

The player's actions also represent a contrast worth review. The current version requires the player to type in characters from the keyboard. The prototype version requires the player to position the cursor and press a button (an item click), an action which is always at the player's fingertips. The two actions are quite different and ease of performance for typing varies significantly from player to player. The

30

```
                        TABLE 1

                STEP 1 - CURRENT VERSION

Determine.the.Directive.Type
  Use 55 for input
  Let prompt.v = "Directive Type:"
  Now Interpret.the.Directive.Meaning
     Given 100
     Yielding meaning
     CC_number = 100
     Find the first Command_Context(CC_Number) in Vocabulary_
     Do until terminated
        Now Determine.the.Response
           Until finished do
              If Input_Line is not empty
                 Use buffer for input
                 Remove first Input_Word from Input_Line
                 Let Response_ = upper.f(IW_Text(Input_Word))
                 Let Last.Source = IW_Source(Input_Word)
                 Destroy Input_Word
                 Return
              Else If Last.Source ≠ 0
                   If Last.Source = I.Terminal
                      Activate Terminal Process
                   Else use 55 for input
                      Now Write.A.Text.String
                         Given prompt.v, 23, 1, 0, 0
                      Use buffer for input
                         Activate Graphics Process
           Suspend
        Loop
     IF Directive_ ≠ 0
        Now Display.a.List
           Given Dir_Menu, 2;
           Let Menu_Status = "display"
           Let Lines = Dim.f(Dir_Menu(*))
           Let lines.per.page = lines-2-2
           If lines ≤ 15
              For I = 1 to lines do
                 Now Write.a.Text.String
                    Given Dir_Menu(I), 2+I-1,1,0,0
                    Call LIBSPut_Screen(Descr.F(Text.String),
                       line+1,column,local.graphics)
              Return
        Let Menu_Status = "menu"
        For each Recognized_Command in CC_SET_OF_ENTRIES(CC_Number)
           With RC_Name = "Sensor List" or RC_Alternate_Name = "SL"
           Find the first case
           Let meaning = RC_Meaning_No  (* = 706 *)
     Return
```

preference of one action over the other varies according to the individual's tastes. These two contrasts are again subtle but their significance, especially typing, is closely tied to an individual's physical skills.

```
                          TABLE 2
                  STEP 1 - PROTOTYPE

ModalDialog(NIL, theItem);
    I1 := GetDItem (theDialog, theItem, Handle, Display);
    I2 := GetNewControl (I1, theDialog);
    DPLongName := GetCTitle (I2, title);
DisposDialog (theDialog);
```

The most significant contrast is in the source of the data. The current version requires the player to be the source of data and it is entered via the keyboard. The prototype version provides the source of data (in all but one case) via computer memory with input made through the item click. Since the input is made through computer memory there is no chance of a syntax error and less of a burden upon the player to enumerate the choices of directive types available to him. The prototype enumerates the choices and makes a syntactically correct entry of data to the process. ·

In summarizing the contrasts of versions in the first step, it is worth noting the amount of source code required to perform the step. The prototype performs the same step with only an estimated 15% of the source code needed in the current version. The primary difference in the amount of code is due to the large amount of current version code required to display, retrieve, and interpret information written to the screen from the keyboard. The prototype version gains this advantage by using operating system functions and procedures to perform comprehensive accomplishment of tasks. Another reason is that fewer tasks are required in the prototype version due to the nature of the operating system. It will be evident as the process continues that the contrasts of display, player actions, and source of data will be factors in each step of the process. The reader is cautioned that task accomplishment may not always be a prototype advantage in the performance of the process. Now step 2 is ready to be done.

2. **Step 2**

This step in the process displays the directive Sensor List on the screen. The display includes the directive title, the directive's attribute and the attribute's field codes. The display is oriented toward the middle of the screen on both versions'

32

TABLE 3

STEP 2 - CURRENT VERSION

```
For each Directive_Prototype with DP_Meaning = 706
  Find the first case
Now Create.the.Directive
  Now Erase.the.Menu.Area
    For I = 3 to 17
      Call LIB$ERASE_LINE(I, 1)
    Let Menu_Status = "blank"
  Create a Directive_
  Store Directive_Prototype in O.DP_Directives_Set
  Reserve Menu Array as size = 15
  Store DP_Menu_Template in Template Array
  For I = 1 to 15
    Let Menu(I) = Template(I)
  Store Menu(1 to 15) in Dir_Menu
  Now Display.a.List
    Given Menu(I), 2
    Let Menu_Status = "display"
    Let lines = Dim.F(Menu(I))
    Let lines.per.page = 18-2-2
    If lines ≤ 15
      For I = 1 to 15
        Now Write.a.Text.String
          Given List(I), 3+I-1, 1, 0, 0
          Call LIB$Put_Screen(Descr.F(Text.String), line+1,
            column, local.graphics)
    Return
  Let Menu_Status = "menu"
```

display layout. In both versions the step begins with a directive meaning and then searches the data structures for a directive prototype having a meaning of "sensor list". When the code finds that data, it displays it on the screen. At this point the versions begin to differ.

The current version first calls a VAX library routine to erase the screen. It then develops a generic display template consisting of 15 lines. Once the template is made, the current version gets each attribute of the directive prototype and draws it to the screen, line by line, according to the AP_Line and AP_Coll values of each attribute. When each line is drawn the display is complete.

The prototype version begins by creating a pointer to a new directive and then invoking the directive display module. Every directive reads in a generic display template and, for each display control item, changes the control's generic title text to the attribute's menu prompt. At the same time each control item is changed, it makes

33

```
                              TABLE 4

                         STEP 2 - PROTOTYPE


For Directive Prototype with DPLongName do
   NewDirective = ↑Directive;
   Directive Display;
       N := NumberAP;   (* the total number of attributes *)
       For I = 1 to 12 do
           GetNewControl(ID, theWindow);
           For J = 1 to N do
               If I = J then
                   SetCTitle(J, AP Prompt);
                   ShowControl (J);
               Else For I > N do
                   For DP Attribute Prototype (I) do
                       HideControl(I);
                       HiliteControl(I, 254);(* disables control *)
           DrawControls(theWindow);
```

it visible. For each control item not changed that control item is made invisible and inactive. The module then draws the display to the screen in its completed form.

The contrasts here are speed and amount of code. The speed is inconsequential here since the prototype uses the same repetitive loop as the current version to produce the display. However, speed could be tilted considerably in favor of the prototype version if each specific directive display existed in a resource file and was explicitly called when needed. This would result in a much faster time for the Macintosh to draw the display to the screen (this will be discussed later in this thesis). The current version has no capability to do this.

The other contrast, amount of source code, again favors the prototype version. The reduction of an estimated 35% of the current version is primarily due to graphics overhead on the VAX. For example, a repetitive loop is used to erase the VAX screen line-by-line, another loop is used to create the display template line-by-line, and finally a repetitive loop is used to draw the display. The prototype version uses a single, doubled-nested repetitive loop to assign text to display items and then draw the display to the screen. The ability of the prototype to do this in a simpler manner than the current version is owed to the operating system functions and procedures comprehensively performing tasks.

34

The summary of contrasts for step 2 again results in a favorable rating of the prototype over the current version. This leads to step 3. Although the prototype version would permit the execution of step 4 at this time, for simplicity in conducting this comparison, both versions will perform the same step.

3. Step 3

This step deals with getting an ID for the directive. Keeping in mind it must be unique, the assumption made here (for simplicity) is that the player will enter a unique ID. In this step both versions differ from the start. Here the prototype version waits for an event to occur while the current version must write the prompt to the screen. The advantage is immediately tilted toward the prototype version. The process will indicate why.

The current version begins by sequentially stepping through the attributes and stops at the first one. It reads the AP_Menu_prompt, "1. List ID:", and rewrites it over the existing "1. List ID" but this time emphasizes it graphically. It then draws the AP_Create_Prompt, with a flashing cursor as emphasis, into the scroll area at the bottom of the screen. Then, invoking the attribute's create routine, it awaits the player's keyboard response. Once the player inputs a string, the current version checks to make sure it is not more than 8 characters. If it is more than 8, an error message is generated and the prompt in the scroll area is rewritten. (This check was intentionally left in the process due to its significance in the comparison.) If the response is acceptable, it is written into the field code space replacing the attribute field code. The current version then deemphasizes the AP menu prompt and then invokes the Retrieve the Directive Attributes module. Here it reads the attribute's word_integer and assigns the ID to the appropriate word in Directive_.

The prototype version begins by waiting for a player action known as an event. In this case, a click in the "1. List ID:" item. The prototype version determines an event occurred, what to do to handle the event, finds out the location of the click event, and then invokes the CRRGet module for the given AP Create Routine. Now the prototype gets a dialog box and draws it in the center of the screen. The dialog box includes a text edit rectangle, 8 spaces long, with a flashing cursor in it. Now the prototype waits for the player to input a string, which is also another event. It determines an event occurred, that it was a keypress event (of a legal character), and enters the character string into the text edit rectangle. Note that since the text edit rectangle is only 8 spaces long it implies the player can never enter a string that is too

35

TABLE 5

STEP 3 - CURRENT VERSION

```
For each Attribute_Prototype in DP_Attributes_Set(Sensor_List)Do
  Use 55 for input
  Let prompt.V = AP_Create_Prompt  (* "List ID:" *)
  Use buffer for input
  Now Write.a.Text.String
     Given AP_Menu_Prompt, AP_Line, AP_Coll, 0, Emphasize_
     Call LIB$Put_Screen(Descr.F(AP_Menu_Prompt), AP_Line, AP_Coll,
     0, Emphasize_)
     Write AP_Arguments_String as text
     Let Subroutine = CRR_Name(AP_Create_Routine)
         Call Get.a.Directive.Identifier
            Read Search.Code
            Until finished do
               Now Determine.a.Response
                  Until finished do
                  If Input_Line is not empty
                     Use buffer for input
                     Remove first Input_Word from Input_Line
                     Let Response_ = Upper.F(IW_Text(Input_Word))
                     Let Last.Source = IW_Source(Input_Word))
                     Destroy Input_Word
                     Return
                  Else
                  If Last.Source ≠ 0
                     If Last.Source = I.Terminal
                        Activate Terminal Process
                     Else use 55 for input
                        Now Write.a.Text.String
                           Given prompt.v, 23, 1, 0, 0
                        Use buffer for input
                           Activate Graphics Process
                  Suspend
               Loop
               If Response_ = "NE"
                  Now write.a.Bottom.Line
                     Given "A null entry is not valid"
                     Call LIB$Set_Cursor(24, 1)
                     Call LIB$Put_Screen(Descr.F(Concat.F(text.string, CR_LF)),
                     24, 1, 0)
                  Now Skip.Rest.Of.Line
                     For each Input_Word in Input_Line do
                        Remove Input_Word from Input_Line
                        Destroy Input_Word
                        Loop
                  Cycle
               Otherwise
               If Length.F(Response_) > 8
                  Now Write.a Bottom.Line
                     Given "ID can be no more then 8 characters"
                     Call LIB$Set_Cursor(24, 1)
                     Call LIB$Put_Screen(Descr.F(Concat.F(text.string, CR_LF)),
                     24, 1, 0)
                  Now Skip.Rest.Of.Line
                     For each Input_Word in Input_Line do
                        Remove Input_Word from Input_Line
                        Destroy Input_Word
                        Loop
                  Cycle
```

36

TABLE 5

STEP 3 - CURRENT VERSION (CONT'D.)

```
If Length.F(Response_) ≤ 8
   For I = 1 to Length.F(Response_) do
      Let This.Char = Substr.F(Response_, I, 1)
      If This.Char = "&" or This.Char = "#"
         Write This.Char as /,
            "Entry cannot contain" This.Char "character"
         Read Output.Line
         Now Write.a Bottom.Line
            Given Output.Line
            Call LIB$Set_Cursor(24, 1)
            Call LIB$Put_Screen(Descr.F(Concat.F(text.string, CR_LF)),
               24, 1, 0)
         Now Skip.Rest.Of.Line
            For each Input_Word. in Input_Line do
               Remove Input_Word from Input_Line
               Destroy Input_Word
            Loop
         Let Bad.Char = 1
         Leave
   Loop
   If Bad.Char = 1
      Bad.Char = 0
   Cycle
      Write Response_ as text
      If Menu_status = "menu"
         Now Write.a.Text.String
            Given Response_, AP_Line, AP_Col2, 8, 0
            Call LIB$Put_Screen (Descr.F(Response),
               AP_Line, AP_Col2, 8, 0)
         Now Replace.the.Menu.Field
            Given Response_
Now Write.a.Text.String
   Given AP_Menu_Prompt, AP_Line, AP_Col1, 0, 0
   Call LIB$Put_Screen(Descr.F(AP_Menu_Prompt), AP_Line,
    AP_Col1, 0, 0)
Now Retrieve.the.Directive.Attributes
   For each Word_Indicator in Word_List(Attribute_Prototype) do
      Case of (WI_Integer)
         (1) Read Dir_ID
      Loop
Loop (* to do next attr'' te *)
```

long and thus never commit an error. The transparent error checking will not accept the player's entry of an illegal character or more than 8 legal characters. The dialog box then assigns the AP field code the value of the string, moves the graphics pen to the field code space on the screen, and draws the string in as the field code. The prototype version then invokes Retrieve the Directive Attributes, reads the attribute's word integer, and assigns the ID to the appropriate word in Directive.

TABLE 6

STEP 3 - PROTOTYPE

```
GetNextEvent(theEvent, everyEvent);
   MouseClick;
      HandleEvent;
         HandleClick;
            K := FindControl(thePoint, theWindow,
             whichControl);
            For DP Attribute Prototype (K) do
               L := AP Create Routine (K);
               CRRGet (L); (* Get an ID *)
                  DialogPtr := GetNewDialog (ID, dStorage,
                   theWindow);
                  TEPtr := TENew (destRect, viewRect);
GetNextEvent;
   Keypress;                 .
      HandleEvent;
         TEKey(Key, TEPtr);
         ModalDialog(theIDFilter, ItemHit);
            theIDFilter(theDialog, theEvent, ItemHit);
               theIDFilter := False;
               ItemHIt : 0;
               If Length.F(string) ≤ 8 then
                  Case theEvent.What of
                     Keydown, Autokey :
                        Case of (theEvent.Message mod 256) of :
                           (1)  (A..Z, 0..9, bkspc) :  ;
                           (2)  NOT (A..Z, 0..9, bkspc) :
                              .       theIDFilter := True;
                                      SystemBeep; .
                                      Return;
                  Else  (* for Length.F(string) > 8 *)
                     If theEvent.Message mod 256 = bkspc then
                        theIDFilter := False;
                     Else the IDFilter := True
                           SystemBeep;
                           Return;
         DisposDialog(theDialog);
         MoveTo(AP Line, AP Col2);   (* AP Line & AP Col2 converted
          to pixel units *)
         DrawString(AP Field Code (K));
         Retrieve Directive Attributes;
            For Q = Word Integer do
               Case of Word Indicator (Q)
                  DirID := AP Field Code(K);   (* Q = 1 *)
```

In this step the contrast focuses on the extra code required by the current version to do the process, the display of the focal point, and ease of input for the player. Once again the advantage pendulum has swung over to the prototype version. The first contrast deals with the fact that the current version is constantly doing graphics tasks of emphasizing, changing, and rewriting. The fact this is done so many

times encumbers the process in the current version while the prototype version performs it's tasks once. The prototype version reduces code execution an estimated 58% from the current version.

A significant contrast is the display focal point. Again the prototype version centers it's focal point immediately grabbing the player's attention. The current version creates two focal points which could be distracting. The first focal point is the emphasized attribute positioned in the mid-upper left portion of the screen. The second focal point is the prompt and cursor down in the screen's scroll area. This is really where the player wants to focus his attention. The advantage regarding this contrast is with the prototype version.

The final point in contrasting the versions is the ease of player input. In the prototype version the player had to select the attribute himself vice having it done for him automatically in a predetermined sequence. This is fairly negligible when compared to the actual entry of data such as the ID string. Here the prototype ensured the player kept the ID within limits while the current version could permit the player to commit an error. This subtle contrast favors the prototype version.

Finally, both versions are ready to enter their list of sensor packages. Considering the wide margin of advantage of the prototype version compared to the current version, the final outcome of the comparison could be predicted. However, step 4 should be examined to complete the process comparison.

4. Step 4

This step is a lengthy process for both versions of source code. Similar processes occur in that both invoke the appropriate create routine, get the sensor package names and display all of them, and invoke Retrieve the Directive Attributes. The similarities end there.

The current version expends a lot of code doing graphics displays, emphasizing, writing prompts, determining responses, rewriting prompts, and deemphasizing. In the end, the current version uses two focal points (moving back and forth between the two points), forces the player to explicitly input whether or not a displayed sensor package is assigned to the list, and requires the player to explicitly close out the list of sensor packages.

The prototype behaves as expected. It waits for an event, in this case the click of the item "2. Sensor", draws a dialog box onto the center of the screen with all the sensor package names included in the box. It also invokes the List Control module

39

TABLE 7

STEP 4 - CURRENT VERSION

```
Use 55 for input
Let prompt.V = AP_Create_Prompt
Use buffer for input
Now Write.a.Text.String
   Given AP_Menu_Prompt, AP_Line, AP_Coll, 0, Emphasize_
   Call LIB$Put_Screen(Descr.F(AP_Menu_Prompt), AP_line, AP_Coll, 0,
   Emphasize_)
Write AP_Arguments_String as text
Let Subroutine = CRR_Name(AP_Create_Routine)
   Call Get.a.Sensor.List
      If Saved.Flag = 0
         Let Saved.Flag = 1
         Create Command_Context
         Let CC_Number = 1120
         Let CC_Message = "Enter (something) or NE to end list."
         For each Sensor_Package
            With SP_Name ≠ " " do
            Create a Recognized_Command
            Let RC_Name = "NE"
            Let RC_Meaning_No = 100
            File Recognized_Command in CC_SET_OF_ENTRIES
            File Command_Context in Vocabulary_
         Store Dir_Menu in Menu (1 to 15)
         Let Menu_Status = "list"
         Let Line = AP_Line + 2
         Let Items.in.List = 0
         Until finished do
            If Line = 18 - 2
               Let Line = AP_Line + 2
            Use 55 for input
            Let prompt.V = "Name of category:"
            Now Interpret.the.Vocabulary.Entry
               Given CC_Number = 1120
            Find Command_Context(1120)
               Until finished do
            Now Determine.the.Response
               Until finished do
                  If Input_Line is not empty
                     Use buffer for input
                     Remove first Input_Word from Input_Line
                     Let Response_ = Upper.F(IW_Text(Input_Word))
                     Let Last.Source = IW_Source(Input_Word))
                     Destroy Input_Word
                     Return
                  Else
                  If Last.Source ≠ 0
                     If Last.Source = I.Terminal
                        Activate Terminal Process
                     Else use 55 for input
                        Now Write.a.Text.String
                           Given prompt.v, 23, 1, 0, 0
                           Use buffer for input
                              Activate Graphics Process
                  Suspend
               Loop
```

40

TABLE 7
STEP 4 - CURRENT VERSION (CONT'D.)

```
If Q = 1
    Let Q = 0
    If Directive_ ≠ 0
        Now Display.a.List
            Given Dir_Menu, 2
            Let Menu_Status = "display"
            Let Lines = Dim.F(List(*))
            Let Lines.per.page = 14
            If Lines ≤ 15
                For I = 1 to Lines
                    Now Write.a.Text.String
                        Given List(I), 3+I-1, 1, 0, 0
                        Call LIB$Put_Screen(Descr.F(List(I)), 3+I-1, 1, 0, 0)
                Return
            Menu_Status = "menu"
            For Recognized_Command if CC_SET_OF_ENTRIES(1120)
            With RC_Name = Response_ or RC_Alternate_Name = Response_
            Find the first case
                Let meaning = RC_Meaning_No
    If meaning = 100
        Leave
    Let Substr.F(Menu_(Line-1), AP_Coll, 15) = SP_Name(meaning)
    Now Write.a.Text.String
        Given SP_Name(meaning), line, AP_Coll, 15, 0
        Call LIB$Put_Screen(Descr.F(SP_Name), line, AP_Coll, 15, 0
    Let Items.in.List = 1
    Create an Element_
    Index(Element_) = meaning
    File Element_ in Vector_
    Add 1 to line
Loop
Reserve Rarray(*) as N.Sensor_Package
Reserve TArray(*) as N.Sensor_Package
Let Categories.per.page = 18-4-3
For each Sensor_Package with SP_Name ≠ " "
    Add 1 to Total.Sensor.Packages.On.This.Side
    Add 1 to Sensor.Packages.On.This.Side
    For each Element_ in Vector_
        With Index(Element_) = Sensor_Package
        Find the first case
        If found
            RArray(Sensor_Package) = 1.0
            TArray(Sensor_Package) = "yes"
        Else
            TArray(Sensor_Package) = "no"
        If Sensor.Packages.This.Side ≤ Categories.Per.Page
            Let Substr.F(Menu(AP_Line+Sensor.Packages.This.Side),
                AP_Coll, 15) = SP_Name
            Let Substr.F(Menu(AP_Line+Sensor.Packages.This.Side),
                AP_Col2, 3) = TArray
Loop
```

41

TABLE 7

STEP 4 - CURRENT VERSION (CONT'D.)

```
Let Header = ITOT.F(N.Sensor_Package)
Write Header as text
For each Sensor_Package
   Write RArray(Sensor_Package)
Release RArray(*)
Now Empty.the.Vector
   For each Element_ in Vector_ do
      Remove Element_ from Vector_
      Destroy Element_
Now Write.a.Text.String
   Given AP_Menu_Prompt, AP_Line, AP_Col1, 0, 0
   Call LIB$Put_Screen
Now Retrieve.the.Directive.Attributes
   Given WI_Integer
   Case of(WI_Integer)
      (53)  Read Dim
         Store 0 in Real_Array(Dim)
         Reserve Real_Array(Dim)
         For I = 1 to Dim
            Read Real_Array(I)
         Store Real_Array(*) in Dir_Generic1_DPointer
         Cycle
```

assigning each package a check box control. It then waits for the player to "check" (an item click) the sensor packages to be listed. The prototype version automatically determines that non-checked sensor packages do not go into the list and automatically closes out the list. It then assigns attributes to the appropriate words in Directive the same as the current version does.

The Sensor List directive is now complete. Again contrasts between the versions gives the prototype the advantage. Amount of code, display focal point, ease of input for the player constitute the areas of difference between the two versions. Amount of code contrasts resulted in the prototype version having an estimated 65% reduction in lines of code executed from the current version. Repetition of graphics code and use of numerous data structure elements not required in the prototype account for the difference and the poor rating of the current version.

The focal point of the display heavily favors the prototype version. It's display is centered on the screen and behaves exactly as the other prototype displays. The current version, on the other hand, continually switched the focal point of the player's attention from mid-screen (to see what was displayed) to the prompt in the scroll area (to make an input). This is distracting and time consuming.

TABLE 8

STEP 4 - PROTOTYPE

```
GetNextEvent
  MouseClick;
    HandleEvent;
      HandleClick;
        K := FindControl(thePoint, theWindow,
        whichControl);
        For DP Attribute Prototype (K) do
          L := AP Create Routine (K);
          CRRGet (L);    (* get a sensor package *)
            List Control(ME);
              DialogPtr := GetNewDialog(ID,
              dStorage, theWindow);
              Until Sensor Package EOF do
                M := SP number;
                SetIText(M, SP Name);
                Repeat;
          For P := 1 to M do
            ModalDialog(NIL, theItem(P));
              I1 := GetDItem(thedialog, theItem,
              Handle, Display);
              I2 := GetNewControl(I1,theDialog);
              AP Field Code (index) := GetCTitle(I2,
              title);
              MoveTo(AP Line + 1 line, AP Col2 +
              4 spaces); (* lines & spaces converted
              to pixel units *)
            DisposDialog(theDialog);
            Index := Index + 1;
            Retrieve Directive Attributes;
              For Q = Word Integer do (* Q = 53 *)
                Case of Word Indicator (Q)
                  For Dim = 1 to M do
                    For Num = 1 to Index do
                      If Sensor Package(Dim) = AP Field
                      Code (Num) then
                        Real Array (Dim) := 1.0;
                        Tarray(Dim) := "Yes";
                      Else
                        Real Array (Dim) := 0.0;
                        TArray := "No";
                    Dir Generic1 DPointer := ↑Real Array;
          For Set of Directives
           with DP Meaning = Sensor List(meaning)
             ↑Set of Directives := NewDirective↑.Directive;
```

The final contrast is ease of use for the player. The player enters data through the keyboard, is subject to committing errors, and must make repeated inputs when using the current version. The prototype version only requires item clicks by the player; no errors, no distractions, just a simple process. The advantage here again heavily favors the prototype version.

In summarizing the entire process, a simplified one at that, the prototype version is distinctly easier to use for the player, a considerable savings of executed source code, and a better presenter of information than the current version. Based upon these comparisons of the two versions the creation of a graphically oriented replacement for the MIP is desireable. The following material is predicated upon the design of such a prototype.

III. THE PROTOTYPE OF THE VISUAL INTERFACE

A. THE METHODOLOGY OF DESIGN

1. The Basics of a Window Management System

Window management systems are relatively new to computer systems, just barely a decade old. The computer industry as a whole did not accept window management from the outset, but people interested in computer graphics have kept the concept alive. Attempts at widespread usage of window management systems failed until Apple introduced their Macintosh. The Macintosh has gained widespread acceptance and is particularly a favorite of casual users. The reason for this is three-fold:

1) Apple specifically concentrated on making the user interface as simple as possible through the use of common visual symbols and association.

2) The company applied an extremely good and technically sound graphics package to the system.

3) Apple took the best ideas of other attempts at developing window management systems and applied them to their design.

As such, the Apple Macintosh has come to be accepted as the "unofficial" standard for the methodology of window management systems [Ref. 6], and it's interface is the foundation of this prototype design.

a. The Infrastructure of the Macintosh Interface

The Macintosh has been described as a universe with its own set of laws, similar to the laws of physics, that describe the standard behavior of objects. These "laws" are consistent which has a direct impact on how an application, and this prototype, is designed. Thus, the application should be flat and user driven (i.e. modeless) as opposed to being tree-structured and menu-driven. This allows the user to focus on what the application does, instead of how it does it. [Ref. 7]

This "modeless" environment allows the user to do anything that makes sense at any time. This means the user is in control of what is going on with the computer and the application. It also means, in general, that if the user performs an action that makes "sense" then the "laws of nature" are not violated and the user doesn't get penalized for doing something wrong. This is a very desireable feature in an application and is the basis for the Macintosh User Interface Standard. [Ref. 8]

The Macintosh User Interface Standard has nine basic concepts:

- Applications   These enable the user to manipulate information.

- Documents   These contain information which the application manipulates.

- Views   These present information.

- Commands   These alter the information in specific ways.

- The Finder's Desktop Metaphor   This provides an image of what is in Macintosh's memory and is a working environment for the information manipulation carried out on the Macintosh.

- Windows   These divide a portion of the Macintosh screen for a portion of the view.

- Selections   These identify those portions of the information that can be affected by certain subsequent commands.

- Editing Conventions   These govern the manner of specifying selections.

- Fonts   These provide a basis for manipulating text appearance.

The reader may gain a comprehensive understanding of the Macintosh User Interface Standard by reading *Inside Macintosh*.

b. The Application of the Interface Standards

These concepts result in the user being presented, on the screen, a variety of graphic objects which behave in expected ways and represent information which the user understands. The user will see at the top of the screen a *menu bar* containing classes of commands. At the user's fingertips is a *mouse* whose movements cause the movements of a *cursor* on the screen. The user can position the cursor over a *menu title*, press the *mouse button* down and, while pressing it down, "pull-down" a list of *menu items*. These cursor movements which "pull down" something are commonly referred to as dragging and have a direct correlation to "dragging" the mouse across a table or desktop. If the mouse button is released over any *menu item*, that item is selected as the command to be performed. The action of pressing and releasing the mouse button is also known as *clicking*. Sometimes these menu items are "dimmed" and cannot be chosen, indicating they cannot be performed at that time by the application.

The user also sees a *window* which presents information such as a document or a message. The window may be "active" and have its objects manipulated. More than one window can be presented at a time but only one may be active. The window

46

presents a view of its contents but not all of its contents may be visible. If so, the user must *scroll* through the information to see it all. The user may move the cursor all around the window and *click* in the window causing something to happen which the user would expect to happen. When finished with the window, the user can click in the *close box* and the window disappears. As with all user actions, the user sees and identifies some object, performs some action with regards to the object, and gets an expected result. This process happens because of the use of a set of Macintosh operating system routines.

These routines are divided by function and are commonly called *managers*. The various managers used in most applications reside in the *operating system* or the *User Interface Toolbox*. The operating system performs such basic tasks as input, output, memory management, and interrupt handling. The user interface toolbox, a level above the operating system, helps implement the standard Macintosh user interface. It is through the variety of managers that the prototype is developed. The managers, all logically named, perform basic tasks as defined by *Inside Macintosh* [Ref. 9]. They are:

- Resource — This manager performs operations on, and allows access to, various application resources such as menus, fonts, icons, windows, etc.

- QuickDraw — The heart of the Macintosh user interface, this manager performs all graphics operations including drawing something on the screen very quickly. It interfaces with many of the other toolbox managers.

- Font Manager — This manager supports the use of the various character fonts when text is drawn by QuickDraw.

- Event Manager — The Event Manager monitors the user's actions and coordinates the actions of the other toolbox routines.

- Window Manager — This manager controls the creation, manipulation and disposal of windows.

- Control Manager — The Control Manager handles special objects on the screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action.

- Menu Manager — This manager creates sets of menus and allows the user to choose from the commands in those menus.

- Text Edit — This manager handles the basic text formatting and editing capabilities in an application.

- Dialog Manager — This manager allows for implementing dialog boxes and the alert mechanism, two means of communication between the application and the end user.

- Desk Manager — The Desk Manager supports the use of desk accessories (mini-applications) in an application.

47

• Scrap Manager	This manager supports cutting and pasting among applications and desk accessories.
• Toolbox Utilities	These are a set of routines and data types that perform generally useful operations such as fixed-point arithmetic, string manipulation, and logical operations on bits.
• Memory Manager	This manager dynamically allocates and releases memory for use by the application and other parts of the operating system.
• File Manager	The File Manager handles file input and output for the operating system.
• Device Manager	The Device Manager manages the input and output devices for the operating system.

2. The Correlation of the MIP and the Macintosh User Interface.

The design of any application must consider the operating system to be used as well as meeting the user's needs. The Macintosh operating system supports the use of high-level programming languages, primarily Pascal. One particularly fast and efficient version of Pascal is Turbo Pascal© by Borland, Inc. Turbo Pascal was this author's choice as the high-level programming language to use for testing some of the concepts of design used in developing this prototype. Turbo Pascal was deemed capable of meeting all requirements of the MIP in terms of functionability.

a. The Restructured Data Flow Diagram

A goal of any application design should be to provide for the smooth flow of data. Figure 2.3 identified a bottleneck of data flow. A distinct advantage of using the visual interface is that this bottleneck can be effectively removed while still leaving the "flow of control" with the user. Figure 3.1 depicts the changes of data flow. The player initiates the flow of data and subsequently controls it all, yet provides little or no data input. The primary difference between this diagram and the first one is that this design use the stored information available to it, primarily the Player Initialization File (PIF), to transform data rather than the player providing the data to be transformed. The application thus runs smoother, information-wise.

A refinement to the data flow diagram explicitly shows how this is supported, Figure 3.2. The transformation *prepare directive* is broken down into three layers. Each transformation's refinement is contained within the dotted lines in Figure 3.2. Note that for each layer the information inputs and outputs are the same respectively. The input to the transformation is the *create* command and the output is the completed directive. In layer two, the transformation is broken down into three transformations which are *get the directive type*, *get the directive attributes* and *assign attribute values*. All the information needed is retrieved from stored information. The

48
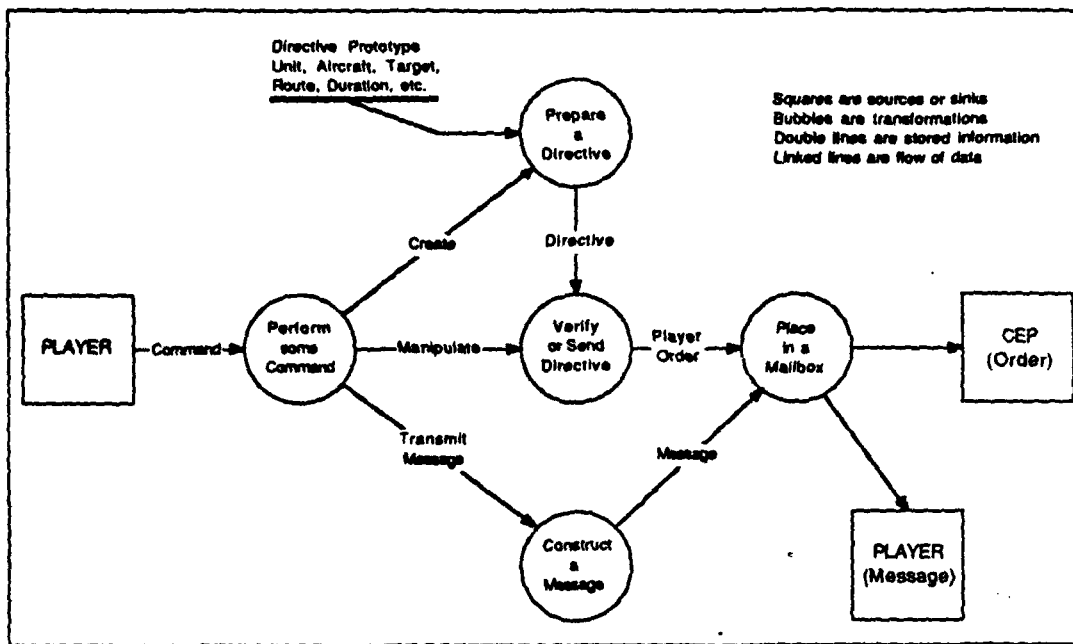
Figure 3.1   The Restructured Data Flow Diagram.
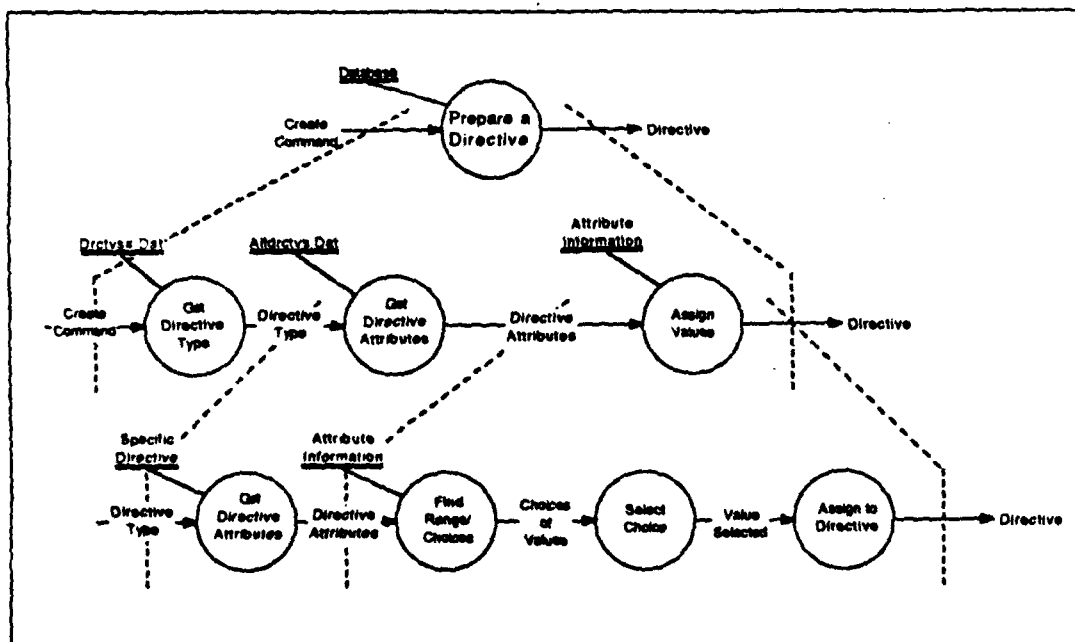
Figure 3.2   The Refined Data Flow Diagram.

49

attribute information is wholly acquired from the PIF. In layer three, refinement of the transformation entitled *assign attribute values* results in the transformations *find ranges/choices available* and *assign choice to attribute*. In following the data flow, the data always exists in the flow pattern and the player controls the data by selecting a choice. The player inputs the *create* command, is presented a list of directives, and selects one. That directive's attributes are presented, and for each attribute the player is presented a range or choice of values from the PIF and selects one to assign as the attribute's value. When all attributes have been assigned values (as required) the directive is complete. It is in this refinement that the MIP becomes an application of the visual interface.

b. The Conversion of SIMSCRIPT to Pascal

(1) *The Data Structures.* The source code of the MIP is lengthy and contains a large number of data structures of the type mentioned earlier in this thesis. There appears to be a strong correlation of these data structures to Pascal data structures. A SIMSCRIPT *set* is equivalent to a Pascal *linked list*. An *entity* is equivalent to a *record* or an *array*, dependent upon the particular entity. In most cases, an entity is of multiple data types so a record is an appropriate structure. An *attribute* is equivalent to a *pointer* or a *variable* of various types (real, double extended, integer, character, enumerated, subrange), or possibly even an array or record. *Temporary entities* are dynamically created during the course of the execution of the MIP, thus they would be created in Pascal as an *addition* to a linked list. *Permanent entities* exist throughout the course of the execution. These are created during initialization and their *size* is known. To correlate this to Pascal, the entities would be *subscripted variables* of an array of records since the size would be the dimension of the array. While SIMSCRIPT automatically provides some pointers and flags to indicate membership in a set or ownership of an entity, these would have to be declared as *fields* of a Pascal record.

An example of this correlation is shown in Figure 3.3. In SIMSCRIPT, the AWACS directive needed all the entities shown in the figure as records. The Pascal version shows the linked lists, the records, and the data fields needed to create the AWACS directive. This correlation can generally be assigned across the board for all the MIP data structures used by the visual interface.

It must be noted that all the data structures used for VAX terminal graphics and for alert/error messages are not needed for the visual interface

50

```
        Records                Variables
Directive  Prototype      Simulation  Time
Attribute  Prototype      Latitude  East
Directive                 Latitude  West
Unit                      Longitude  North
Aircraft                  Longitude  South
Emitter  Suite            Create  Routine
Sensor Package
```

NOTE: This does not include graphics data.

Figure 3.3   The AWACS Directive PASCAL Data Structures.

application. The data contained in these structures is necessary due to the operating system used. The data structures necessary for the Macintosh operating system are inherent to it or can be explicitly addressed in the code or resource files. Specifically, the entities are *interval_, database_, interrupt_, input_word, element_, command_context, recognized_command, CEP_parameter_index, long_word, menu_line,* and *held_directive.* There are also several variables not needed which generally pertain to terminal graphics. Should any question arise about the purpose of and necessity for any SIMSCRIPT data structures to be used in the visual interface, the reader should refer to the source code and the MIP Software Engineering Maintenance Manual, Reference 10.

(2) *The Source Code.* SIMSCRIPT II.5© was designed to support structured programming and modularity as applications of software engineering [Ref. 1: p. i]. As such, many of the coding conventions of the SIMSCRIPT language are similar to the language constructs of the high-level programming languages such as Pascal. The *if-then-else, do-while,* and *case* statements are examples of condition statements common to both languages.

Reserve, define, mode, and *dimension* are typical assignment statements in SIMSCRIPT, but must be handled through Pascal declaration and assignment statements accordingly. The use of boolean arguments is common to both languages

51

and the operators are the same. A brief review of the SIMSCRIPT reference would allow any programmer with just moderate Pascal experience the ability to read and follow the source code.

c. The Prototype of the MIP Functions

(1) *The Commands.*   The commands have a direct functional correlation to *menu items* in the Macintosh environment. The commands can be grouped by class in nearly all cases. The classes of commands are implemented as *menu titles.* The few which are not associated closely with a particular class have been loosely grouped together into a class entitled *special.* In one case, a single command was categorized as a class itself. This was the *find* command. Several commands are also inherent to the Macintosh operating system. An example of this is the *hold* command. It is equivalent to the Macintosh *open* command.

The specific menu items and their functional definitions are as follows:

- About...   *About...* prompts the display of a window containing information about this prototype.

- Desk Accessory   This command calls the specified desk accessory to begin operation (normally on-screen) for the user.

- Create   This calls a procedure to create an action or utility directive.

- Open   *Open,* an operating system feature, is similar to the MIP's *hold* command; it opens an existing file or document.

- Save   This operating system feature stores a named file.

- Save as...   This operating system feature stores a file after prompting for and receiving a filename from the user.

- Close   This operating system feature closes an open file. The user is given a choice, if necessary, of saving changes or not.

- Print   This operating system feature prints the open file at the Macintosh printer.

- Send   *Send* calls the send procedure to send a directive to the game.

- Verify   This command calls the verify procedure to ensure a directive is OK to send to the game.

- Quit   This operating system feature quits executing the application when selected.

- Group Create   This command calls a procedure to create a group of directives.

- Leave   This command calls a procedure to take a directive out of a group.

- Join   This command adds a directive to a group.

- Group Send   This command sends a group of directives to the game.

- Time Increment   This command calls the IncGroupTime procedure to increment the execution time of the groups directives by a certain amount.

52

- Transmit Message   This command allows a player to prepare and send a message to another player or a group of players.

- Receive Message   This command allows a player to read his messages which are in the message queue.

- Query   The *query* command lets the player request a report from the game.

- Graphics   This command permits the player to make adjustments to his graphics station during the course of the game.

- Find   This is a class of commands to find a specific group, directive, message or report which the player may have filed away.

- Edit   This class of commands is composed entirely of operating system commands which the player uses to edit text, etc.

- 'Trash'   This command permits the deletion of any file by "dragging" that item to the trash can so the can is highlighted.

These commands are incorporated into the Macintosh application by pre-coding them into a *resource file*. The resource file is read by the application program and the *menu bar* is constructed from the data contained there. Subsequently, any time a menu item is selected by a player, the command is carried out by the program. An interesting feature of the menu items (and an entire menu list), is that they can be enabled or disabled as required during the course of execution of the program. Certain MIP commands cannot be performed while other actions are being performed by the player. The Macintosh program can handle these situations by disabling the necessary commands in the menus.

The maintenance of the menu items must be done in three places, dependent upon the maintenance required. The resource file must be updated, the menu resource declaration in the program may need to be updated, and the source code to handle the menu event must reflect the changes made, as required. While this maintenance, on paper, seems elaborate, in practice it is relatively simple in most cases and will probably be rare as the addition or deletion of commands is not anticipated for the game itself. The source code may change as procedures called from the commands are added or deleted.

(2) *The Directives.* The directives correlation to the prototype is that they represent information to be manipulated. Manipulation of information is done via the *window*. Hence, each directive is displayed in a window. The directive attributes are displayed as information with predefined positions in the window. It is possible (and very probable) that they will not all be visible to the player. The player will have to scroll to view the attributes remaining out of view.

53

The method of assigning values to an attribute is consistent and straight forward. The player moves the cursor over an attribute and clicks the mouse button. This *event* activates a procedure which draws a *dialog window*. The contents of the dialog window are dependent upon the range or choice of values which can be assigned to that attribute as it's value. *Controls* are used to make the assignments. If a number is needed, a control called a *dial control* is used with minimum and maximum values representing the range of values for that attribute. If a string is needed, such as a choice from a list, the list is displayed and each item has it's own *button control*. The player selects the choice and that choice is assigned as the attribute value.

The attribute values all represent some data base information stored in the individual player's functional game file called the Player Initialization File (PIF). The PIF gets created by the game director during game preparation and represents the only correct and authorized set of information in any given game scenario. By using the PIF, the range/choice of values can be determined based on the conditions of directive type, units and their missions, unit resources, resource characteristics, and environmental data. It should be stressed that direct usage of the PIF data to populate "pop-up" windows or dialog windows is very efficient and not now being done in the current version. By reading the given PIF data into the dialog window, a value can be selected by the player. This is a significant change since the player no longer has to thumb through a sometimes large "player manual" to choose data and then correctly enter that data via the keyboard. The player can see that data in front of him, comprehend it quickly, and "enter" the data in syntactically correct format; all by clicking a button!

This process is repeated many times during the course of a game and is in keeping with the refined data flow diagram design, Figure 3.2. It is natural, consistent, and permissive (for the most part) - three fundamentals in designing a visual interface such as this prototype for the Macintosh [Ref. 9: p. I-27]. In the following sections, the prototype is established as an application and the reader will be able to see how the aforementioned processes are implemented in an application such as the MIP.

54

B. THE PROTOTYPE - MACMIP

1. The Prototype Abstraction

The prototype, appropriately named MacMIP, was developed to provide the JCS managers another way to use and play JTLS. This prototype takes on a different appearance than most programs. Macintosh documentation stresses the point that Macintosh programs like MacMIP "don't quite look the way they do on other systems." In Apple's words, "Everything you know is wrong." [Ref. 9: pp. 1-4, 4(Draft)]

The reason is simply that event-driven programs behave differently and have a different structure. The first-level abstraction of MacMIP, Figure 3.4, shows the set-up of the program.

```
Program MacMIP  (Input, Output);

Declarations
      Libraries;
      Constants;
      Types;
      Variables;

Utility Functions and Procedures;

Menu Driven Functions and Procedures;

Event Driven Functions and Procedures;

Initialization Functions and Procedures;

Cleanup Functions and Procedures;

Main Program.
```
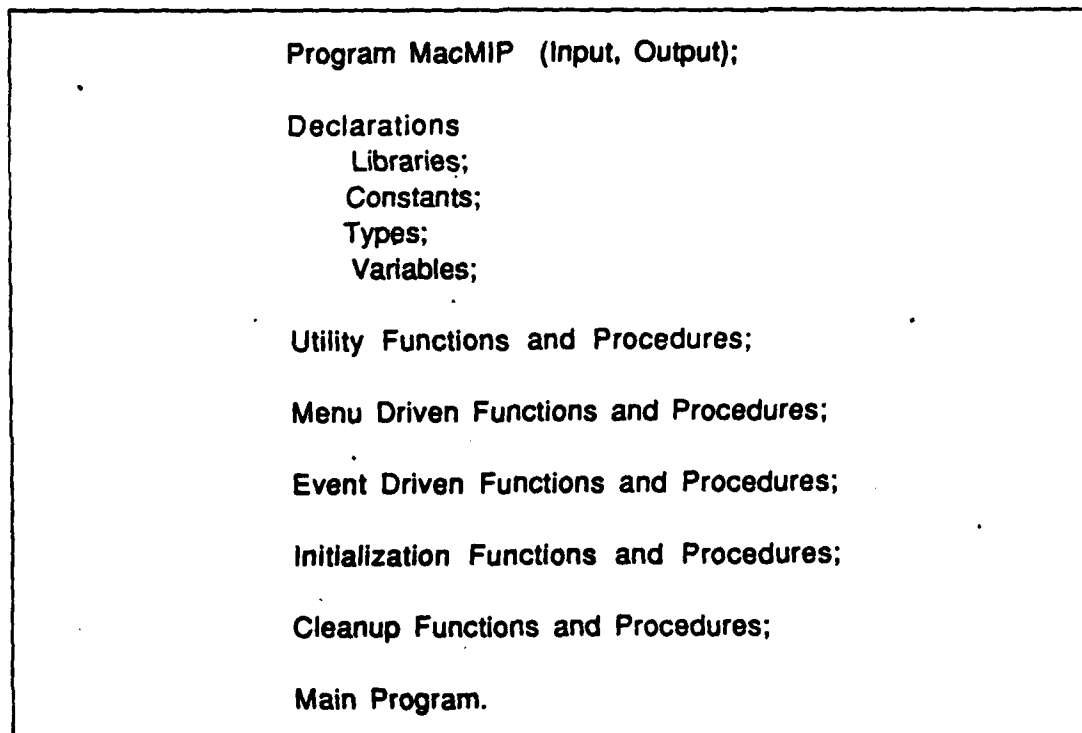
Figure 3.4   An Overview of MacMIP's Program Structure.

In the material that follows, the first-level abstraction is refined into an abstraction level that is suitable for use as a guide for coding the prototype. The refined abstraction is a third-level abstraction and it's elements are categorized and

their purpose defined. In fact, the third-level abstraction was used to code the prototype version used in the code comparison section of Chapter 2. The results of that code comparison demonstrate the desirability of continuing with the development of the prototype from an efficiency standpoint. A complete version of the third-level abstraction may be found in appendix A of this thesis.

a. The Header and Declarations

The *header* is typical of any program. It simply invokes the program. The *declarations* section is again typical. It identifies operating system libraries used, establishes global variables by type, assigns values to constants (including the identification of resource files used), and formally sets up the data structures.

The first significant difference from most programs is in the declaration of procedures. These application-specific procedures are categorically grouped together. The categories are *utility*, *menu-driven*, *event-handling*, *initialization*, and *cleanup*.

(1) *Utility Functions and Procedures.* The utility category is generally used as a catchall for the functions and procedures which do not belong to any other category. The utility functions and procedures and their purposes are as follows:

- Directive — This procedure draws a specific directive onto the screen. It is called when a directive type has been specified.

- Attribute Display — This procedure highlights a directive attribute, calls a dialog box and displays the attributes range choice of values for selection. returns the selected value, and assigns it to the attribute. It is activated by an event.

- Assignments — This is a set of procedures which match up to directive types. They handle anomalies in the process of assigning directive attribute values to particular fields of player · orders. These are necessary since the MIP does not have a generic algorithm to do this.

- Verify — This is also a set of procedures which match up to directive types. Each verifies that the specific directive attributes are correct (outside of standard type-checking) and are assigned to the appropriate field in the directive record. They are called by the Send, Verify, Group Send, and Group Verify command procedures.

- Retrieve Attribute Values — This simply assigns attribute values to specific directive fields.

- Player Order Assignment — This procedure creates a player order record by assigning a directive's attribute values into specific player order fields in order to "match up" to the CEP's equivalent of a player order record. To handle the anomalies of any specific directive, the procedure calls the necessary *assignment* procedure. Player Order Assignment is called by the Send. Group Send, Query, and Transmit Message commands procedures.

56

- Mail a Player Order/Message

  This concatenates the player order or message into a string for purposes of sending it as an ASCII file to the CEP. It is called by Player Order Assignment.

- Quick Order Display

  This procedure behaves similarly to Directive Display. It is called by the Query and Graphics Adjust command procedures.

- Quick Attribute Display

  This behaves similarly to Attribute Display.

- List Control

  This procedure takes a list, assigns each list member a control, and then draws the control into a dialog box. It is called by numerous procedures.

- Time Dial Control

  This procedure creates a dial control with range values commensurate with a minimum and maximum time, and draws the control into a dialog box. It returns a time value.

- Lat/Long Dial Control

  This procedure creates a dial control, with minimum and maximum values, and draws it into a dialog box. It returns a latitude longitude point.

- Integer Dial Control

  This procedure creates a dial control, with minimum and maximum values, and draws it into a dialog box. It returns an integer value.

- Real Dial Control

  This procedure creates a dial control, with minimum and maximum values, and draws it into a dialog box. It returns a real value.

- Lat/Long Conversion

  This function takes the starting geographic point(SW) and the number of x,y hexes and determines the NE point of the playing surface. It then converts that point to lat/long coordinates for use as game boundaries.

- Read From VAX

  This procedure is used to communicate with the VAX by receiving.

- Write To VAX

  This procedure is used to communicate to the VAX by writing to it.

- PIF Update

  This is used to update a wide variety of the player's database when MacMip is used interactively during game play.

- Write The Status

  This procedure writes and updates that status dialog window. It is called by PIF Update.

- Write the Player

  This procedure writes and updates the player dialog window. It is called primarily during initialization.

- CRR Get

  This is a set of procedures which get lists, times, geographic points, altitudes, etc. Each procedure has a direct correlation to the MIP source code. They are called by numerous higher-level procedures.

(2) *Menu-Driven Functions and Procedures.* The **menu-driven** category is a collection of functions and procedures which are called as a result of a player selecting a menu item. These functions and procedures may in turn call a host of utility and

57

operating system functions and procedures. In general, these are called only from the *Handle Menu* procedure in response to an event. They are essentially used to carry out MIP commands which are not handled by the operating system. The menu-driven functions and procedures are:

- Do — This procedure simply draws a dialog box whose contents are information about MacMIP. It calls nothing.

- Desk Accessory — This procedure starts up a specified desk accessory for the player's use.

- Create — This procedure issues the command to create a new directive. It calls a dialog window so the player may select a directive type. When the type is chosen, the Directive Display procedure is called.

- Send — This procedure prepares actions directives for "mailing" and then places the orders into the mailboxes. It calls Verify, Player Order, and Mail a Player Order/Message.

- Verify — This procedure calls a directive specific verify procedure.

- Group Create — This procedure establishes a group into which directives may be collected.

- Join — This procedure assigns an existing directive to a group.

- Leave — This procedure removes a specific directive from a group.

- Group Send — This procedure prepares a group of directives for "mailing" to the CEP and then places them in the mailbox. It calls the Verify and Send procedures for each directive in the group.

- Group Verify — This procedure verifies that each directive in a group can be sent to the game. It calls the Verify procedure for each directive in the group.

- Group Time Increment — This procedure increments the time of execution for each directive in a group. It calls the CRR Get and the Time Dial Control procedures.

- Transmit Message — This procedure allows the player to create and send a text message to another player or players. It calls the Mail a Player Order/Message procedure.

- Receive Message — This procedure retrieves a message from the player's message queue and displays it on the screen so the player may view it.

- Query — This procedure, similar to Create, requests reports from the game when MacMIP is in the interactive mode of operation. It calls the Quick Order Display procedure.

- Graphics Adjust — This procedure, also similar to Create, makes adjustments to the player's game graphics stations. It calls the Quick Order Display procedure.

- Find — These procedures find any particular group, action directive, utility directive, message, or report that the player has filed away.

58

(3) *Event-Driven Functions and Procedures.* The event-driven procedures are those procedures which are performed as the result of the occurrence of some event. An event is normally a mouse click or a keystroke performed by the player. System events, such as the movement of the mouse, are also members of this category. Parameters of the events are examined to determine what occurred, where it occurred, and what is supposed to happen next. In turn, the event-driven procedures are invoked to handle the event. In a sense, these procedures are the "brains and nerve center" behind the application's "bodily functions." The procedures are:

- Mouse   This procedure identifies where the mouse was clicked and then calls another event to handle the task to be done as a result of the click and it's location.

- Keypress   This procedure handles a keystroke event, including command keys. It may or may not explicitly call other event procedures.

- Update   This procedure handles updates to the three windows of MacMIP. It calls various procedures dependent upon what needs to be updated.

- Handle Menu   This procedure handles the event of a click in a menu item and calls the necessary menu-driven procedure including operating system procedures.

- Cursor Adjust   This procedure changes cursors based upon the cursor's screen position as a result of mouse movement.

- Handle Event   This procedure determines what event occurred and oversees the performance of the task to be done as a result of the event.

(4) *Initialization and Cleanup Procedures.* The Initialization and cleanup procedures are the *start* and *stop* of the program. The initialization procedure initializes everything in the program at the start of the program's execution. It could be constructed as a combination of several procedures but here it has been treated as a single giant procedure with calls to a few utility procedures. The cleanup procedure is invoked at the termination of the game. It simply erases the contents of the screen, logs off the VAX, and shuts down the Macintosh. The initialization and cleanup procedures are each invoked once during the execution of MacMIP.

b. The Main Body of MacMIP

The main body of MacMIP is a short, concise set of statements which are the "soul" of the Macintosh. After initialization, the program performs a repeating loop until told to quit. The loop first checks to see if any systems-defined tasks need to be done. If so, the Macintosh does them. The loop then checks to see if any events are in the event queue. If so, the system gets the first event of the highest priority class and performs the event-driven task. It then repeats the loop. When the system is told to quit, it invokes *Cleanup* and erases the screen.

59

2. Background Issues of Prototype Design

There were several issues which were (and still are) challenges to fully implementing MacMIP. The challenges primarily of interest here are the transition of an application (the MIP) from one computer system to another of a different format and the physical data link between the different systems. These challenges are not impassible but they do warrant special mention so the reader understands the task at hand.

The task of implementing a prompt-based program, designed and written for a VAX minicomputer, into a graphics-oriented, event-driven operating system such as the Macintosh provides several challenges. First, it is not a trivial process since the Macintosh applications do not carry out a sequence of steps in a predetermined order. Rather, the Macintosh program is driven by user actions (such as clicking and typing) whose order of occurrence cannot be predicted. Thus, the SIMSCRIPT program cannot be running parallel to the Macintosh and expect the Macintosh to emulate a VAX terminal and still function in the visual interface mode. MacMIP must be programmed to account for the occurrence of events; the MIP's prompt-based applications are not event-driven.

Secondly, a thorough concept of graphics capabilities is necessary to effectively apply the visual interface to the MIP through the Macintosh operating system. Prompt-based applications such as the MIP generally use "menus" to display the prompts. Moving through the prompts is done sequentially due to the application's rigid tree-like hierarchical structure; one prompt must be answered correctly before another one can be dealt with, especially if it resides on another level of the hierarchy. The code to set-up the prompts and move between them is usually rigidly structured as well. The Macintosh system does all this through the use of it's graphics toolbox *QuickDraw* and the *Resource Manager*. Thus, any MIP source code dealing with the CRT display is totally unusable in MacMIP. To try to transfer it to the Macintosh would require too much source code just to negate those CRT instructions. Simply stated, reformatting is not trivial.

The other issue of establishing a link to the VAX is also one which is possible but not trivial. Although the exact mechanics of establishing the link will not be dealt with here, it must be noted that the capability to link the Macintosh to the VAX has been demonstrated by the Jet Propulsion Laboratory, the Naval Postgraduate School's C3 Laboratory, and the Warrior Preparation Center, among others. The reason for mentioning it is that the Macintosh runs only one application at a time. Therefore, the

instructions to link to the VAX on an interactive basis must be incorporated into MacMIP's source code. It is also likely that the JTLS Executive Level Program must know that a particular link and game mailbox is a Macintosh and not a VAX VT-100 terminal. With these issues in mind, the general format and design of the prototype can be implemented.

C.   CREATING DIRECTIVES WITH MACMIP

The use of MacMIP to create the AWACS directive will result in the same directive being created as examined earlier in this thesis. The method of creating it now has a new look. To appreciate this prototype, the reader is invited to step through the process again.

The process begins with the player. With the mouse at his fingertips, the player moves the cursor around on the screen. As the cursor moves across the menu bar, the player positions the cursor over one of the menu titles and presses the mouse button. While holding down the mouse button, the player "pulls down" a list of menu items by dragging the mouse, Figure 3.5. As the cursor passes over the pulled-down menu items, the player releases the mouse button while the cursor is positioned over the create command. This constitutes an event so the Macintosh software determines what event occurred and where, and, having recognized the event, handles it. In doing so, the menu-driven procedure *Create* is invoked.

The issuance of a command starts the ball rolling. First, MacMip reads a resource file to get a dialog window, gets a list of directives the player can create, invokes list control, and finally draws all of them onto the screen, Figure 3.6. The player can quickly and easily see what his options are. The player can select a specific directive type or cancel the *create* and quit. If the player cancels, nothing happens except the command is canceled. Actually, the player can quit at any time without penalty; the main window is simply erased. If the player selects a directive type such as AWACS, Directive Display is invoked.

When Directive Display is invoked, MacMIP reads a resource file which places control buttons in a pre-determined order, assigns an attribute name to each button, and draws the AWACS attributes onto the contents region of the main window. As the player clicks on any attribute, the attribute control is enabled and invokes the attribute display procedure.

Apple Menu	File Menu	Edit Menu	Group Menu	Special Menu	Find Menu
About...	Create	Undo	Create	Tx Msg	Group
Desk Accessory	Open	Cut	Join	Rx Msg	Directive
	Save	Copy	Leave	Query	Utility
	Save as...	Paste	Send	Graphics	Messsages
	Close	Clear	Verify		Reports
	Print	Select All	Time Inc		
	Send	Duplicate			
	Verify				
	Quit				

Figure 3.5   The MacMIP Menu Bar with Menu Items.

Now MacMIP determines the type of attributes (squadron, for example) clicked on and determines the range of values or choices eligible to be assigned as the attribute's value. In the case of squadron, this would be a list of air squadrons with an AWACS mission. MacMIP then draws a dialog box, with the appropriate controls and information, Figure 3.7, and waits for the user to select a value. Once this is done, MacMIP assigns the value to the attribute. For example, 73 AWACS SQ would be selected as the value of the attribute squadron. The dialog box is erased and the portion of the directive covered by the dialog box is redrawn.

When the player selects an attribute which is a utility directive, such as Air Route, the player has the option of referencing the air route ID or creating the air route directive. If the first option is selected, MacMIP behaves as normally expected for an attribute and displays a list of choices. One choice is an empty textedit box so the player can reference a yet to be created Air Route. If the player chooses the latter option, MacMIP remembers the AWACS window, invokes Directive Display again, given a type of "air route", and draws a new window over the AWACS window. NOTE: This is similar to the OVERMIP feature of the MIP but this is not restricted to just three windows or limited performance of commands. With a new window, MacMIP can perform any command allowed for an active window and it's function,

```
 File   Edit   Group File   Special   Find
┌─────────────────────────────────────────────┐
│              The Status Line                 │
│              The Player Line                 │
│ □ ▬▬▬▬▬▬▬▬▬  Directive Type: ▬▬▬▬▬▬▬▬         │
│                                              │
│  1. MISSION:        xxxxxxxx    8. ORBIT LAT/LON:  dd-mm-ssD ddd-mm-ssD
│                                              │
│  2. SQUADRON:       xxxxxxxxx   9. SENSOR LIST:    xxxxxxxx │
│  3. * AIRCRAFT:     nnnn                      │
│                                              │
│  4. ROUTE IN:       (xxxxxxxx)               │
│  5. ROUTE OUT:      (xxxxxxx)                │
│                                              │
│  6. ORBIT ENTRY TIME:  ddhhmmZMMMMYY         │
│  7. ORBIT DURATION:    ddDhhHmmM             │
│                                              │
└─────────────────────────────────────────────┘
```
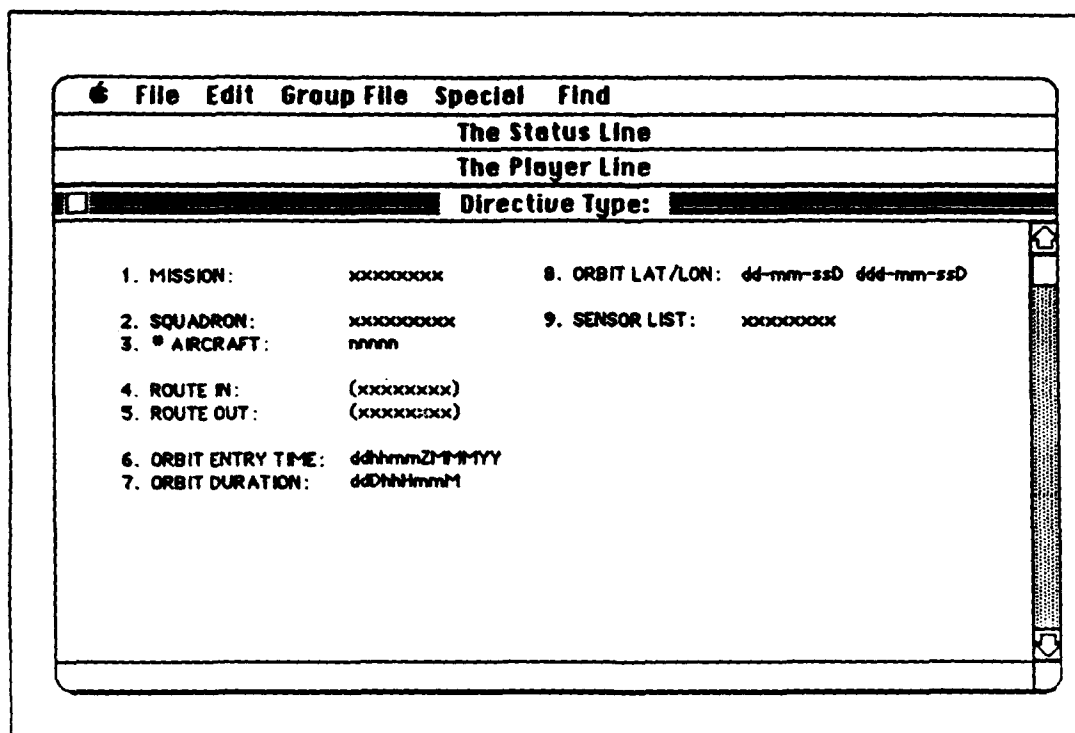
Figure 3.6   The AWACS Directive as drawn by MacMIP.

regardless of the number of windows. Practical limitations of approximately 12 windows are dictated by Macintosh operating software but the only physical limitation is with memory space on the system. Now Air Route is handled just like any other directive. When the player is done with Air Route, he simply saves it. The Air Route window is erased while it's information is stored somewhere in memory. The player is now returned to the AWACS directive window and continues to process information in it.

This process continues on for the player at his will. He would never have to hold a printout on his lap to find data. It would always be available to him on his computer "desktop." The player would control the data yet quickly move between different tasks of varying modes as he deemed necessary and never lose any "document." It would always be somewhere on his desktop!

63
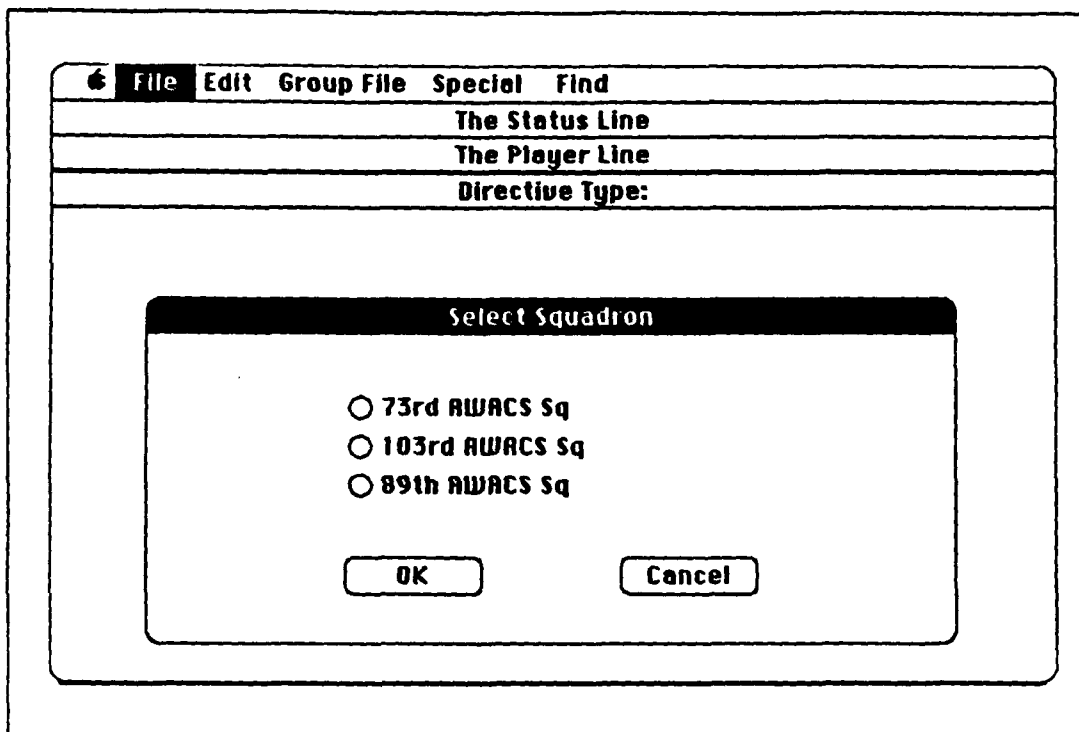
Figure 3.7   The Directive with the Dialog Box.

D.   THE FUTURE UTILIZATION OF THE PROTOTYPE

1. Technical Aspects of Prototype Utilization

A brief review of the methodology used is in order to shape the prototype's future.   The methodology used to develop this prototype was simple and straightforward.   The ultimate goal was established as the application of the Model Interface Program onto the Macintosh operating system.   The process of doing this can be mapped out in steps.   First, understand the MIP operations, i.e. what it does. Then understand the SIMSCRIPT program language and how it operates on the VAX minicomputer.   A collateral task is to understand the Macintosh operating system, it's capabilities, and the programs it uses well.   Then the task is to understand the design and structure of the MIP and correlate it into a design using the visual interface.   Once this is done, the next phase is to translate the design concepts into a code-like format so the prototype takes on a realistic look.   This is the point where the prototype development is now.

64

In getting the prototype to this point, much of the original source code was examined to determine how the MIP works. In doing so, it became evident that much of the logic and algorithms used would be effective in MacMIP. The reason is that the "behind the screen" manipulation of information by the MIP is fairly effective so there is no reason to re-invent the wheel. It is the format and presentation of the data which sparked the idea for the prototype in the first place. With this in mind, it becomes self-serving to use that code in this prototype. This is evident by the references made to specific MIP modules in the MacMIP psuedocode. An underlying premise is that the development and production of MacMIP would be considerably shortened compared to a full re-design.

There were numerous ideas borne out of this development with regards to future prototype development. One idea mentioned earlier was that of placing individual directives into resource files. This would speed-up Macintosh operations and provide à cleaner, sleeker display. The better the graphics, the better the visual interface. The MIP currently reads in all commands and all of the various directives, queries, adjustments, etc., from a database. The database is not expected to change significantly over time so maintenance and currency should not be significantly impacted upon. While the MIP currently reads the data in based upon player function, the same school of thought could apply to MacMIP. The answer is to have a separate diskette per function and simply load that function's diskette into the Macintosh when that function is used. One advantage to this is to effectively utilize memory space. Another advantage, for game management, will be addressed shortly.

A second idea, which follows the lead of the first, is to place each player's PIF on a separate diskette as well. The PIF is developed by the CEP upon initialization of a scenario database. Since the PIF doesn't change unless the scenario does, it is feasible to pre-load the PIF for each scenario used. A separate diskette per function per scenario would allow great flexibility in the use of the prototype. An extension of this advantage would be that only the function affected by a change to a scenario would have to be updated. This idea would also save machine memory space, a concept which closely relates to the way JTLS already reduces CPU input/output by using video disc digital graphics.

Another feature of the MIP which has not been addressed in the prototype is the system capability for the expert player. Presently the expert player can type all the directive data into one string in a predetermined order (this is called stacking), enter it,

65

and have a complete directive. This capability is a natural for a prompt-based application. However, with the Macintosh and the visual interface format in use, the stacking capability is a diametrical opposite. As such, it was not designed into the prototype. To fully realize the potential of MacMIP, this capability should be incorporated into MacMIP as a text edit faculty.

Finally, an aspect of development to be considered is a total re-design of the MIP. The key issue with the MIP is it's ability to communicate the player's intentions to the game. This is done by passing ASCII data between the two programs. Therefore, the MIP could manipulate it's data in many different ways just as long as the file passed was in proper order and format. Consider first that SIMSCRIPT is a modeling programming language. The MIP per se models nothing. It is written in SIMSCRIPT to be consistent with the other JTLS programs. Instead of being a model which generates data, the MIP simply manipulates data. Since the MIP manipulates data, consider the possibility that there is a more efficient method of manipulating that data. That method is a data base management system (DBMS). Several excellent systems exist which were designed expressly for the Macintosh. One of these or a specially designed system might do a better job of dynamically manipulating the large amounts of data used in JTLS. If a decision was made to use a very capable workstation, such as the SUN or IRIS workstations, for a future generation JTLS input device, a DBMS system coupled with a windowing and resident color graphics environment becomes a very attractive system option. A single workstation could easily function as a graphics station and MIP substitute.

2. Managerial Aspects of Prototype Utilization

JTLS was originally developed with military training in mind. As events have transpired that original premise has been overcome. The issue of computer simulations used as planning aids has come to the forefront. With the proliferation of the desktop microcomputer, prototypes like MacMIP take on increased importance. One important reason is found in the methods used by planners to test various strategies and tactics. A planner develops a strategy and then must test it for feasibility. If the planner could prepare in advance all of the missions expected to be used for a given strategy, then the planner could do all that work in his office where all his references, working papers, etc., are located. When the planner tests his plan, it is done in the computer laboratory. Using a portable system of diskettes from a desktop computer, the directives could be transported (so could the Macintosh for that matter) to the lab

66

and loaded into the game. This would save a considerable amount of time for the planner as the game could be played faster, more repetitions could be run with more variations of game parameters, and a greater spectrum of outcomes realized for analysis of plan effectiveness.

A reason of secondary importance is found in the basic premise of the visual interface. It is geared toward the casual user. The military planner is not a computer systems expert by trade. The planner's expertise can range from the novice category to the expert. By designing and using a system like the Macintosh, with it's visual interface, the needs of all users can be met. One can assume that even the expert is not likely to use JTLS on an everyday basis over an extended period of time. With so much diversity in a planner's work, it would be easy for even the expert JTLS gamesman to lose his grasp of the game's nuances. With a continual change of scenarios, the data used by the player would change and further compound the problem of maintaining game skills. The prototype would quickly return the planner to a high level of effectiveness in game skills, or quickly train the planner new to JTLS, due to it's graphical orientation and it's ease of use. If correctly designed it will also reduce input error rates at all stages of training of the player-analyst or player-planner.

IV. CONCLUSIONS

The original purpose of this thesis was to examine enhancement of player inputs to JTLS through computer graphics techniques. The overall result of the examination is that a graphical application of the game is a very efficient and a desirable method to effect player inputs. This result is supported by positive use of human visual information transfer, the ability of computer software such as window management systems to convey this information, and the capabilities of hardware such as the Macintosh operating system. Symbolic association has long been recognized as a positive method of communication. The use of computer graphics is a logical extension of that school of thought and has found an application in window management systems. The windowing capabilities in the Macintosh, when compared to the prompt-based VAX, show a distinct advantage in providing ease of player input and, at the same time, indicates a potential savings in the amount of code necessary to perform the same operations on the VAX. The results of this examination fully supported the development of the prototype.

The prototype design shows how to improve the current methods of effecting player inputs. The design of the prototype incorporates the advantages mentioned above. The design identifies the areas of the Model Interface Program most in need of enhancement and then breaks down the functions of each area by correlating them into visual (graphical) objects. The design also identifies a very capable language (Pascal) for coding such a prototype and correlates the original SIMSCRIPT source code (data structures, logic, and language constructs) to it.

This road map of design leads directly to the pseudocode abstractions. These abstractions show that the coding of the prototype is possible and goes so far as to lay out the program's skeletal structure. The categorization of MIP functions allows for explicit definitions and routines of MacMIP which in turn perform the MIP's functions. The road map allows for a total rewrite of the MIP. The next step to be taken in the design process is to actually begin coding. Although a total rewrite is a large undertaking, and beyond the scope of this thesis, it is the most efficient and economical method of implementing the graphical enhancements.

68

In the case of the Macintosh, the powerful capabilities of the microcomputer would be lost if it was coded to simply emulate a VAX VT-100 terminal. Then the Macintosh graphics would not provide any true enhancements to the player input mechanism. Also, while the psuedocode was written with the Macintosh in mind, it is purported to be general enough to provide decisionmakers a basis for which to apply the MIP functions to other graphics-oriented window management computer workstations. Indeed, the proliferation of low-cost microcomputers with graphics capabilities give the prototype increased credibility.

In summary, the prototype can be a valuable tool to JTLS managers in the near future. The design is generic enough to apply to any window management system but is ready to be coded for the Macintosh. The best of the original source code has been applied to the prototype to aid quick implementation. It's use in a desktop, office environment will provide the manager greater flexibility in utilizing JTLS to it's full capability and worth.

APPENDIX
MACMIP: THIRD-LEVEL ABSTRACTION

```
******************************* HEADER *********************************

Program MacMIP (Input,Output)

************************** DECLARATIONS (of the globals) ***************************

*** Operating System Functions ***

        (#R ± )                *range checking, on/off
        (#I ± )                *input/output error checking, on/off
        (#I <filename>)        *inclusion of file(s)
        (#B ± )                *bundle bit, on/off
        (#R <filename>.Rsrc)   *identify resource files used
        (#T APPLDMO1)          *set application identification
        (#U ± )                *auto-link to runtime units, on/off
        (#S ± )                *use of segmented code, on/off
        (#S <segment name>)    *name of segmented code used

*** Macintosh Interface Units ***

USES
        PasInOut       *Implements the standard Pascal input/output (I/O) routines.

        MemTypes       *Defines special Macintosh data types and must be in any Mac-style
                        application.

        QuickDraw      *The Macintosh graphics package.

        SCSIIntf       *Provides access to interface port and permits communications with
                        the port.

        OSIntf         *The operating systems interface which performs lowest level basic
                        tasks.

        TooIntf        *Implements the user interface features of windows, menus,
                        controls, dialog boxes, text editing commands, etc., and must be
                        in any Mac-style application.

        PackIntf       *The interface to packages of data structures and routines which
                        are stored as resources.

        MacPrint       *Provides access to Macintosh printing manager.

(** NOT USED:  Any of the following may actually be needed when MacMIP is actually coded,
however,  they  do  not  appear  necessary  at  this  time:   PasControl, PasPrinter, SANE,
FixMath, Graf3d, AppleTalk, SpeechIntf.**)


************************** Constants, Types, and Variables *************************


CONSTANTS
        Menu List Count = 6           *total number of menus
        Apple Menu = xx               *the resource
        File Menu =xx                 *ID unique
        Group File Menu =xx           *to each
        Edit Menu                     *specific
        Special Menu = xx             *menu file
        Find Menu =xx

        AM = x                        *index
```

70

```
        FM = x
        GM = x                          *into
        EM = x                          *menu list
        SM = x                          *for
        FDM = x                         *each
                                        *menu

        Main Window ID = xxxx
        Status Window ID = xxxx                    *the resourceID
        Player Window ID = xxxx                    *unique to each
        Attribute Window ID = xxxx                 *specific
                                                   *window used
        Buffer Size = xxx
        Buffer Count = xxx                         *for disk I/O
                                                   *for disk I/O
TYPE
        Player File = RECORD  of
                        PF Concat.F : char
                        PF Suffix   : char
                        PF Unit     : integer

        Player = RECORD of
                        PL side                 :  char
                        PL side number          :  integer
                        PL function             :  char
                        PL function no.         :  integer
                        PL receives input       :  integer
                        PL graphics station     :  integer

        Mailbox  =  RECORD  of
                        MBX logical name   :  char
                        MBX size           :  integer
                        MBX channel.       :  integer

        Message  =  RECORD  of
                        MSG status   :  integer
                        MSG text     :  char

        Unit  =  RECORD  of
                        UT Pointer
                        UT long name                :  integer
                        UT short name               :  string
                        UT type                     :  string
                        UT AS aircraft available    :  integer
                        UT AS aircraft type         :  integer
                        UT side                     :  integer
                                                    :  integer

        Directive  =  RECORD  of
                        DIR ID
                        DIR unit 1              :  char
                        DIR unit 2              :  char
                        DIR unit 3              :  char
                        DIR unit 4              :  char
                        DIR target 1            :  char
                        DIR target 2            :  char
                        DIR lat 1               :  char
                        DIR lat 1 text          :  double extended
                        DIR lon 1               :  char
                        DIR lon 1 text          :  double extended
                        DIR lat 2               :  char
                        DIR lat 2 text          :  double extended
                        DIR lon 2               :  char
                        DIR lon 2 text          :  double extended
                        DIR time                :  char
                        DIR time text           :  double extended
                        DIR duration            :  char
                        DIR duration text       :  double extended
                        DIR generic 1 text      :  char
                        DIR generic 2 text      :  char
                        DIR generic 3 text      :  char
```

71

```
                    DIR generic 1 integer    :  integer
                    DIR generic 2 integer    :  integer
                    DIR generic 3 integer    :  integer
                    DIR generic 4 integer    :  integer
                    DIR generic 1 double     :  double extended
                    DIR generic 2 double     :  double extended
                    DIR generic 3 double     :  double extended
                    DIR generic 4 double     :  double extended
                    DIR generic 1 Ipointer   :  integer
                    DIR generic 1 Tpointer   :  integer
                    DIR generic 1 Dpointer   :  integer

    Attribute Prototype  =  RECORD  of
                    AP prompt
                    AP field code           :  string
                    AP arguments string     :  string
                    AP number               :  string
                                            :  integer

    Directive Prototype  =  RECORD  of
                    DP long name
                    DP short name           :  string
                    DP meaning              :  string
                    DP CEP class            :  integer
                    DP assignment routine   :  integer
                    DP verify routine       :  integer
                    DP attribute prototype  :  integer
                    DP number attributes    :  array (1 to 12) of RECORD
                                            :  integer

    Quick Attribute Prototype  =  RECORD  of
                    QAP prompt
                    QAP create routine      :  string
                    QAP arguments string    :  integer
                    QAP conversion type     :  string
                    QAP PO word             :  integer
                    QAP all flag            :  integer
                                            :  integer

    Quick Order Prototype  =  RECORD  of
                    QOP context
                    QOP full name           :  integer
                    QOP numeric name        :  string
                    QOP CEP class           :  string
                    QOP CEP specific type   :  integer
                    QOP CEP number          :  integer
                    QOP message             :  integer
                    QOPQAP                  :  integer
                                            :  array (1 to 4) of RECORD

    Air Weapon  =  RECORD  of
                    AW name
                    AW X air ground         :  string
                    AW X weapon weight      :  real
                    AW X supply category    :  real
                    AW X night factor       :  real
                    AW X weather factor     :  real
                    AW X weapon color       :  real
                    AW X weapon effects     :  real
                    AW X long range         :  real
                    AW X precision guided   :  real
                    AW X weapon speed       :  real
                                            :  real

    Aircraft  =  RECORD  of
                    AC name
                    AC X range              :  string
                    AC X day night          :  real
                    AC X crew time          :  real
                    AC X fuel               :  real
                    AC X weather factor     :  real
                    AC X runway required    :  real
                    AC X type               :  real
                    AC X wet weight         :  real
                                            :  real
```

72

```
                         AC X dry weight         :  real
                         AN X EC factor          :  real
                         AC X max altitude       :  real
                         AC X speed              :  real
                         AC X load time          :  real
                         AC X aircraft side      :  real
                         AC X damage ratio       :  real
                         AC X refuel             :  real
                         AC X spare              :  real
                         AC X enemy detection    :  real
                         AC X engage fuel        :  real
                         AN X AI range           :  real

Supply Side, Supply Category  =  RECORD  of
                         SS name      :  string
                         SS units     :  string
                         SS multiplier :  real

Function  =  RECORD  of  FN name  :  string

Unit Type  =  RECORD  of  UTP TEXT  :  string

Target Type  =  RECORD  of  TTP TEXT  :  string

Emitter Suite  =  RECORD  of  ES name  :  string

Word Indicator  =  RECORD of Word Integer  :  integer

Sensor Package  =  RECORD  of  SP name   :  string
                                SP number :  integer

CRRGet Routine  =  RECORD of Create Routine  :  integer

Associated Directive  =  RECORD  of  AD dir ID  :  string

Month  =  RECORD  of
                   Mth name   :  string
                   Mth length :  integer

Order Record  =  RECORD  of
                   OR time text            :  string
                   OR time                 :  real
                   OR DP meaning           :  integer
                   OR status               :  string
                   OR Message              :  string
                   Associated Directive    :  RECORD

Player Order  =  RECORD  of
                   PO class                :  integer
                   PO specific type        :  integer
                   PO unit                 :  integer
                   PO time effective       :  real
                   PO word 1 integer       :  integer
                   PO word 2 integer       :  integer
                   PO word 3 integer       :  integer
                   PO word 4 integer       :  integer
                   PO word 5 integer       :  integer
                   PO array 1 pointer      :  integer
                   PO array 2 pointer      :  integer
                   PO array 3 pointer      :  integer
                   PO array 4 pointer      :  integer
                   PO array 5 pointer      :  integer
                   PO word 1 real          :  double extended
                   PO word 2 real          :  double extended
                   PO word 3 real          :  double extended
                   PO word 4 real          :  double extended
                   PO word 5 real          :  double extended
                   PO word 1 text          :  char
                   PO word 2 text          :  char
```

73

```
Target  =  RECORD  of
                TR pointer   :   integer
                TR number    :   string
                TR name      :   string
                TR type      :   integer

VARIABLES
        Date Stamp,
        Status Line,
        Special Status,
        Player Line,
        User ID,
        Simulation Time Text,
        Scenario Name,
        Game Classification,
        Supply Field        :   char;

        AMP Flag,
        Starting Day,
        Starting Month,
        Starting Year,
        No of Login Builds,
        Air Refuel Index,
        Supply Width,
        Supply Precision,
        Sun Status,
        Number X Hexes,
        Number Y Hexes      :   integer;

        Simulation Time,
        Supply Minimum,
        Supply Maximum,
        Lat Hex One,
        Long Hex One
        Lat East
        Lat West
        Long North
        Long South          :   real;
```

*********************** Utility Functions and Procedures ***************************

*These subroutines may or may not be representative of original MIP logic. If there is a
correlation, then the MIP subroutine ID will be annotated within brackets to guide the
programmer to that original source code.*

Directive Display;

*This display is called from a resource file, develops a specific directive's display, and
draws it into the main window.*

        Given a DP meaning;
        Read the resource file for the generic directive display;
        For each static item do;
                index I from 1 to 12;
        Get attribute prototype of DP meaning;
                index J from 1 to N;   *N is the number of attributes
                                                 for the given DP meaning.
                For I = J do;
                        Change the static text to represent AP prompt and AP field code;
                For each I > N do;
                        Hide the static item so it can't be seen or enabled;
        Now draw the display into the main window;

End Directive Display;

Attribute Display;

74
```

*This is activated when a directive attribute is highlighted. It determines what type of
attribute it is, gets the appropriate type of control, gets a range/choice of eligible
values, and assigns each of them a control. It then draws the control into a dialog box
onto the screen. When a particular control is activated, that value is assigned to the
attribute variable.*

        Given AP(J) with a DP Meaning;
        Read a resource file for the dialog box;
        For AP create routine (J) do CRRGet;     [I100]
                Case of (1 to 25);  *get the appropriate create control routine
        Draw the dialog box with the eligible values and their controls;
                For activated dialog box assign a value to word;
                        *Of Word Integer of AP(J).

End Attribute Display;


Assignment;

*This procedure handles the anomalies found when assigning directive-specific attributes
to the Player Order fields.  All AIR directives (DP meaning 301 to 399) use assignment 300
as well as their own specific assignments.*

                Given a DP meaning;
        Invoke assignment.DP meaning;  [A101 to A104, A106, A123, A200 to A227, A300, A306,
                                        A312 to A314, A401 to A407, A501 to A503, A800]

End Assignment;


Verify;

*This procedure verifies that certain directive-specific assignments were made before
allowing the Player Order to be sent to the CEP.  The first part of the assignment checks
for the existence of a referenced utility directive and the remainder invokes the
directive-specific verifications.*

        The send flag is not set;
        Given a DP meaning;
        For each utility directive in directive;
                Determine it exists:
                        No  :  Draw an error dialog box to alert the player;
                        Yes :  Then go on;
        If utility directive is weapon load;
                Determine weight of load for Aircraft is OK:
                        No  :  Draw an error dialog box to alert the player;
                        Yes :  Then go on;
        Invoke verify.DP meaning;  [V101, V104, V106, V123, V200, V202 to V209, V211, V214,
                                    V217, V218, V222, V225, V300, V306, V312, V401 to V407,
                                    V501, V800]

End Verify;


Retrieve Directive Attributes;  [U019]

*This procedure assigns a blank string to attributes with "Null Entry" as values.  This
permits acceptable formatting of the Player Order.*

        Given word integer;
        Case of that word;

End Retrieve Directive Attributes;


Player Order Assignment;  [A000]

*This procedure assigns directive common attributes to specified Player Order fields.
Given a DP meaning it then invokes that directives specified assignment.*

        Given Directive with DP meaning;
        PO class = DP CEP class;
        PO specific type = DP meaning;
        PO time effective = DIR time;
        PO word 2 text = DIR ID;
        If DIR unit 1 name ≠ Null Entry;
                For UT short name = DIR unit 1 name;
                Let PO unit = UT pointer;
        Invoke Assignment.DP meaning;

End Player Order Assignment;


Mail the Player Order/Message;  [U018]

*This procedure concatenates the Player Order into one string of char for purposes of
sending it to the CEP. After the Player Order is made, it is read to port. This is done
by invoking Write To VAX. Then the Player Order is reset to 0.*

        Given a Player Order;
        For directive with DP meaning = 101;
                Increment No of Login Builds;
        Convert all integer and real values to character;
        Concatenate all Player Order fields into one string;  *This is known as a message.
        Determine that the message will fit into the mailbox:
                No  :  draw an error dialog box to alert the player;
                Yes : If message is directive then;
                        OR time text = Simulation time text;
                        OR DP meaning = DP meaning;
                        OR unit = DIR unit 1 name;
                        AD DIR ID = DIR ID;
                ·File Order Record;
        If mode is "on-line" put the Player Order message into the mailbox;
        Reset Player Order = 0;  *For the next time.

End Mail the Player Order/Message;


Quick Order Display;

*This procedure is similar to Directive Display but on a smaller scale.*

        Given a QOP context;
        Read the resource file for the generic display;
        For each static text item do;
                Index I = 1 to 6;
        Get Quick Attribute Prototype for QOP context;
                Index J = 1 to N;  *N is the number of attributes.
                For I = J do;
                        Change the static text to represent QAP prompt;
                For I > N do;
                        Hide the static text so it can't be seen or enabled;
        Now draw the display into the main window;

End Quick Order Display;


Quick Attribute Display;

*This is similar to Attribute Display but on a smaller scale.*

        Given QAP(J) with QOP context;
        Read the resource file for the dialog box;
        For QAP Create Routine (J) do CRRGet:
                Case of (1 to 25) : Do Create Control;  *As appropriate.
        Draw dialog box with eligible values and controls;

76

End Quick Attribute Prototype;


List Control;

*This procedure develops and draw a dialog box with controls for each item in the list.
It returns a value of char.*

    Given N number of items in list;
    If List is multiple entry then
            Assign a check box for each item;
    Else
            Assign a radio button for each item;
    Map item to graphics position;
    Draw the dialog box in the main window;

End List Control;


Time Dial Control;

*This procedure develops and draws a dialog box with a scroll dial to return a value of
time.  Minimum time always = 1 minute.*
    Given maximum day value;
    Given game minimum time;  *If time is for "Duration" then game minimum time = 0
                                since the value needed is a block of time for incre-
                                mental purposes.
    Determine minimum and maximum values for the control:
            Minimum value = game minimum time + 1;
            Maximum value = game maximum time + maximum days;
    Draw a scroll dial using minimum/maximum values;
            Current value is minimum value;
            Format is 2 places for days, 2 places for hours, 2 places for minutes;
    Return a time value;  *Usually is converted to a real number in terms of days.

End Time Dial Control;


Lat/Long Dial Control;

*This procedure develops and draw a dial control onto the dialog box to return geographic
points.  Only degree fields must have values.  Direction value in text converts ± for
real values of lat/long.*

    Given a minimum and maximum value;  *Usually the game boundaries.
    Determine the number N of points to be made;
    Draw a scroll dial with minimum/maximum values;
            Current value is minimum value;
            Format is 3 places degrees, 2 places minutes, 2 places seconds, and 1 place
              direction;  *For lat the first place has a value of 0.
    For N points do;
            Enter a latitude;
            Enter a longitude;
            Convert all points to real values;

End Lat/Long Dial Control;


Integer Dial Control;

*This procedure develops and draws a dial control in a dialog box and returns an integer
value.*

    Given a minimum and a maximum value;
    Draw a scroll dial with minimum/maximum values;
            Current value is minimum value;
            Format is 5 places;
    Return an integer value;

77

End Integer Dial Control;

Real Dial Control;

*This develops and draws a dial control with minimum and maximum values and returns a real value.*

    Given minimum and maximum values;
            Current value = minimum value;
            Format is 9 places with 5 decimal places;
    Draw a scroll dial with minimum and maximum values;
    Return a real value;

End Real Dial Control;


Lat/Long Conversion;[U013, U014, U015, U016]

*This develops the game surface boundaries in terms of latitude/longitude for use as dial ranges.*

    Given Lat Hex One, Lon Hex One, Number Y Hexes, and Number X Hexes;
    Convert to Lat Hex X, Lon Hex Y;
    Convert hexes to coordinates;
            Results in Latitude East/West and Longitude North/South;

End Lat/Lon Conversion;


Read From VAX;

*This uses the RS-232 port as a device and reads an ASCII file from the device if something is in the buffer. The buffer must be checked periodically.*

End Read From VAX;


Write To VAX;

*This writes an ASCII file to the RS-232 port when called to do so by the Macintosh. It contains protocol information for the VAX and Macintosh to communicate.*

End Write To VAX;


PIF Update; [CEP Process]

*This reads a message from the CEP, determines it's type and subtype, and take the necessary actions depending upon the type.*

    Read From VAX;
    For message type case of :
            (1)  Message  :  do
                    increment queue by 1;
                    file message in queue;
                    Write The Status given queue;
            (2)  Time  :  do Write The Status given time;
            (5)  Interrupt pending  :  do Write The Status given special status;
            (6)  Target  :  do a new target record;
            (7)  Game speed  :  do Write The Status given game speed;
            (8)  Stop password  :  change the password;
            (9)  PIF updates  :  case of subtype  :
                    (1)  Aircraft available  :  find the unit, change it's number;
                    (2)  Cargo trucks available  :  find the unit, change it's number;
                    (3)  Tanker trucks available  :  Find the unit, change it's number;
                    (4)  Aircraft characteristics  :  change the aircraft's record;
                        [U028]
                    (5)  Air weapon characteristics  :  change the air weapon's record;

78

[U029]
(6) Personnel weight  :  change the personnel logistics load record;
(7) Aircraft name   :   change the aircraft's name;
(8) Air weapon name   :   change the air weapon's name;
(9) Sensor package name   :   change the sensor package's name;
(10) Emitter package name   :   change the emitter package's name;
(11) Supply category  :  change a supply category's record;
(10) Sunrise-sunset  :  do Write The Status given sunstatus;

End PIF Update;


Write The Status

*This procedure develops and draws the status window and the information it contains.*

Given queue, game speed, starting time and date, and special status;
Read a resource file to get a dialog window;
For each static text, change the static text to the necessary value;
Draw the dialog box;

End Write The Status;


Write The Player;

*This procedure develops and draw the player window and the information it contains.*

Given classification, player function, scenario name;
Read a resource file to get a dialog window;
For each static text, change the static text to the necessary value;
Draw the dialog box;      .

End Write The Player;


CRRGet Procedures;

*These procedures get certain information for attributes and directives.  Each gets some specific information, thus they are listed along with their MIP module number.*

Get an ID  [U101]
Get a Duration  [U110]
Get a Lat and Long  [U111]
Get a New Target Name  [U106]
Get a New Target Number  [U105]
Get a Real Number  [U114]
Get a Route  [U115]
Get Additional Route Info  [U026]
Get a Runway Name  [U124]
Get a Sensor List  [U118]
Get a Supply Changes List  [U122]
Get a Supply Load  [U116]
Get a Target Name  [U104]
Get a Target Types List  [U123]
Get a Targets List  [U113]
Get a Time  [U109]
Get a Unit Name  [U103]
Get a Units List  [U112]
Get a Weapon Load  [U117]
Get an Emitter List  [U119]
Get an Integer  [U112]

End CRRGet;


*********************** Menu Driven Functions and Procedures ***********************

*** Apple Menu Functions and Procedures ***

Do About;

*This procedure simply tells the user some information about MacMIP.*

    Read the information needed from resource files;
    Put together a string of parameter text;
    Get the dialog box and put it up;
    When it has been read, get rid of it;

End Do About;


Do Desk Accessory;

*This gets the selected desk accessory and starts it up.*

    Save a port for the desk accessory;
    Get the desk accessory name;
    Start the desk accessory;

End Do Desk Accessory;


*** File Menu Functions and Procedures ***

Create;  [Command(1) of C000]

*This procedure determines what directive to build  and does it.*

    Determine the directive type (Action or Utility);  [C001]
            For type selected put up a dialog window with list of DP meanings;
                    Given list do List Control;
            Return the selected value = DP meaning;
            Dispose of that window;
    Given DP Meaning do Display procedure;  [C002]
            For each attribute highlighted do Attribute Display procedure;  *Get a value
                                                                      for    the
attribute.
            Do Retrieve Directive Attributes procedure;

End Create;


Send;  [Command(101) of C000]

*Sends only action directives, not previously sent, to CEP.  The file sent must be open in
window.*

    Given DP meaning do Verify procedure;
            Check verify flag case of:
                    Not set  : verify and set flag;
                    Set  :  go on;
    Do Player Order Assignment;
    Mail the Player Order/Message;

End Send;


Do Verify;  [Command(106) of C000]

    Check verify flag case of :
            Set  :  Tell player that directive is OK;
            Not set  :  For DP meaning do Verify procedure;

End Verify;


80

*** Group File Menu Functions and Procedures ***

Group Create;  [Command(51) of C000]

*This procedure creates a group of directives.*

        Get a unique ID for the group;
        List the action directives which do not already belong to a group;  [C103]
                Do List Control;
                Return a value;
                Place the selected directive in the group;

End Group Create;


Join;  [Command(109) of C000]

*This procedure adds a directive to a group.*

        Given an open directive put it into an existing group;  [C103]

End Join;


Leave;  [Command(110) of C000]

*This procedure removes a directive from a group.*

        Given an open group and a list of it's directives;  [C104]
                Do List Control
                Returned value is selected directive;
        Remove selected directive from group;  *It will stand alone.

End Leave;


Group Send;  [Command(54) of C000]

*This procedure sends a group to the CEP by sending a directive at a time.*

        Check group to ensure Directives with DP meanings 310 and 800 aren't in the group at
          the same time;  [C009]
                Yes  :remove one of them;
                No   :  then for each DP meaning do Verify;
        When all are verified do for each :  Send;  *One at a time.

End Group Send;


Group Verify;  [Command(58) of C000]

*This procedure verifies a group of directives.*

        Do Verify procedure for each directive in group;  [C009]
                Except if DP meaning = 800;  *Verification not done on Air Mission Package.
        Set verified flag;

End Group Verify;


Group Time Increment;  [Command(59) of C000]

*This procedure creates a block of time to add to a group.*

        Check group directives for DIR Time Text $\neq$ 0 and DIR Time Text $\neq$ Null Entry;
          [C010]
                If so for that directive(s) increment DIR Time and DIR Time Text;
                        Do Time Dial Control;
                        Return a block of time;

81

End Group Time Increment;


*** Special Menu Functions and Procedures ***


Transmit Message;  [Command(5) of C000]

*This procedure allows the player to create a text message to send to another player.*

      Select Player(s) to send message to;  [C010]
           Includes "all", "all Blue", "all Red";
           From is entered as Player's function and side;
      Enter message as a text string;
           Enter "//" To indicate the end of the message;
      Invoke Player Order Assignment;
           Invoke Mail the Player Order/Message;
           Do as a repeating loop to place into each mailbox as required;

End Transmit Message;


Receive a Message;  [Command(14) of C000]

*This procedure is invoked only when their is a CEP message in the queue.  It pulls out
the CEP message on a FIFO basis for the player to read or print.*

      Read message file for first message;  [C006]
      Draw that message to the main window;

End Receive a Message;


Query;  [Command(13) of C000]

*This procedure creates request for the CEP to send a progress report to the player.  It
is similar to creating a quick order.*

      For QOP context = 90 invoke Quick Order Display;  [C017]
           Do List Control to return the type of report;
      Given a report type do Quick Attribute Display;
      Do Player Order Assignment;
      Do Mail the Player Order/Message;

End Query;


Graphics Adjust;

*This procedure makes adjustments to the player's graphics station.*

      For QOP context = 98 invoke Quick Order Display;  [C017]
           Do List Control returning the type of adjustment;
      Given an adjustment type invoke Quick Attribute Display;
      Do player Order Assignment;
      Do Mail the Player Order/Message;

End Graphics Adjust;


*** Find Menu Functions and Procedures ***


Find Group;

*This procedure invokes the operating system finder given the type of "group".*

End Find Group;


82

Find Directive;

*This procedure invokes the operating system finder given the type of "type of directive".*

End Find Directive;


Find Utility;

*This procedure invokes the operating system finder given the type of "type of utility directive".*

End Find Utility;


Find Message;

*This procedure invokes the operating system finder given a type "filed messages".*

End Find Message;


Find Report;

*This procedure invokes the operating system finder given a type "filed reports".*

End Find Report;


*********************** Event Driven Functions and Procedures ***********************


Mouse Click;

*This identifies where the mouse was clicked and invokes the necessary procedure.*

```
      For location case of :     *Somewhere in the main window.
                Menu bar   :   Do Handle Menu;
                Content    :   Do Handle Click  *In the main window to handle the attributes.
                Close box  :   Do Handle Close
                System window  :  Do System Click  *This is a click in a desk accessory.
```

End Mouse Click;

Keypress;

This handles the event of any keystroke including the use of command keys.

End Keypress;

Update;

This invokes the necessary update procedure depending upon the event.

```
      Case of :
                Status window  :   Do Write The Status;
                Player window  :   Do Write The Player;
                Main window   :
                        Get the new information to go into the contents;
                        Erase the current contents;
                        Draw the new contents in it's place;
```

End Update;

Handle Event;

83

This determines what event occurred and handles it.

```
        Case of  :
                Mouse Click      :   Do Mouse Click;
                Key down         :   Do Keypress;
                Autokey          :   Do Keypress;
                Update event     :   Do Update;
                Activate event   :   Do Activate;
```

End Handle Event;

Cursor Adjust;

*This changes cursors based upon the location of the cursor on the screen. These are
application specific changes, not those made by the operating system. These may not be
need for MacMIP but is included here just in case.*

        Do change to cross, arrow, plus;

End Cursor Adjust;

Handle Menu;

*This procedure handles the event of any menu item being hit and invokes the necessary
action to take place.*

```
        Case Menu of :
                Apple Menu  :  Case of  :
                        About             :  Do About;
                        Desk Accessory    :  Do Desk Accessory;
        File Menu  :  Case of  :
                        Create       :   Do Create;
                        Open         :   operating system feature;
                        Save         :   operating system feature;
                        Save As...   :   operating system feature;
                        Close        :   operating system feature;
                        Print        :   operating system feature;
                        Send         :   Do Send;
                        Verify       :   Do Verify;
                        Quit         :   operating system feature;
                Group Menu  :  Case of  :
                        Create           :  Do Group Create;
                        Leave            :  Do Leave ;
                        Join             :  Do Join;
                        Send             :  Do Group Send;
                        Verify           :  Do Group Verify;
                        Time Increment   :  Do Increment Group Time;
                Special Menu  :  Case of  :
                        Transmit Message  :  Do Transmit Message;
                        Receive Message   :  Do Receive Message;
                        Query             :  Do Query;
                        Graphics          :  Do Graphics Adjust;
                Find Menu  :  Case of  :
                        Group        :  Do Find Group;
                        Directive    :  Do Find Directive;
                        Utility      :  Do Find Utility;
                        Message      :  Do Find Message;
                        Report       :  Do Find Report;
                Edit Menu  :  Case of  :
                        Undo         :   operating system feature;
                        Cut          :   operating system feature;
                        Copy         :   operating system feature;
                        Paste        :   operating system feature;
                        Clear        :   operating system feature;
                        Select All   :   operating system feature;
                        Clear        :   operating system feature;
```

84

End Handle Menu;

******************** Initialization Functions and Procedures ********************

Initialization;

*This procedure initializes the various Macintosh managers, variables, and procedures. It
also initializes the PIF by reading in the appropriate database information.*

Initialize;
     GrafPort, Font, Window, Menu, Text, Dialog, Events Managers.
     Cursors;
     Menus;
     Windows;
     Variables;
             Supply Field = "nnnnnnnn.nn";
             Supply Min = 0.0;
             Supply Max = 999999999.99999;
             Supply Width = 15;
             Supply Precision = 5;
             Month (1 to 12) = Jan..Dec;
             Month Length (1 to 12) = 31..31;
     Load database information; [I000]
             Write to VAX;
                     "Open [.data]miscel.dat"
                     "Read [.data]miscel.dat"
             Read From VAX;
             Store data into appropriate files;
             Write To VAX;
                     "Close [.data]miscel.dat";
                     "Open [.data]executive.dat";   [I001]
                     "Read [.data]executive.dat";
             Read From VAX;
             Write To VAX;
                     "Close [.data]executive.dat";
                     "Open [.data]drctvs<function number>.dat"; [I003]
                     "Read [.data]drctvs<function number>.dat";
             Read From Vax;
             Write To VAX;
                     "Close [.data]drctvs<function number>.dat";
                     "Open [.data]alldrctvs.dat";   [I003]
                     "Read [.data]alldrctvs.dat';
             Read From VAX;
             Write To VAX;
                     "Close [.data]alldrctvs.dat;
                     "Open [.data]quick<function number>.dat";
                     "Read [.data]quick<function number<.dat"; [I003]
             Read From VAX;
             Write To VAX;
                     "Close [.data]quick<function number>.dat;
     For MIP mode = "on-line";
             Write To Vax;
             Call VAX 'SYS$CREMBX';   *Create the mailboxes.[I302]
             Read From VAX;   *Get the mailbox names.
     Load the PIF; [I003]
             Write To VAX;
                     Given function number and type of game = "start";
                     "Open PF_Unit(file_type)";
             Read From VAX;
                     Read game class, starting day, starting month, starting year;
                     Latitude/Longitude information;
                     Unit names,types, and resource information;
                     Target names, types, numbers;
                     Supply side/category names, units, multipliers;
                     Aircraft information;
                     Air weapon information;

85

```
                        Sensor package information;
                        Emitter suite information;
                Write To VAX:
                        Close PF_unit(file_type);

End Initialization;


********************************** CLEANUP **********************************

Cleanup;

*This procedure simply erases the screen when the game is finished, logs off the VAX, and
shuts down the Macintosh.*

End Cleanup;


********************************** MAIN **********************************

BEGIN MacMIP;

        Call Initialization;
        Repeat until finished;
                System Task;  *For desk accessories.
                Cursor Adjust;
                If GetNextEvent(everyEvent,theEvent);  *If there is an event...
                        Then Handle Event(theEvent);  *...then do it.
        Call Cleanup;  *When finished.

END MacMIP;
```

LIST OF REFERENCES

1. Russell, E.C., *Building Simulation Models with SIMSCRIPT II.5*, CACI, Inc.-Federal, Los Angeles, California, 1983.

2. Contingency Planning Subtask, Joint Theater-Level Simulation, *Executive Overview*, Jet Propulsion Laboratory, Pasadena, California, 1986.

3. Fredericksen, M.N., *Aiding Computer Application Programmers and Users with the Tools of the Visual Interface*, M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1986.

4. House,W.C., *Interactive Computer Graphics Systems*, Petrocelli Books, New York, New York, 1982.

5. London, K.R., *The People Side of Systems*, McGraw-Hill, London, England, 1976

6. Hopgood, F.R.A., and others, *Methodology of Window Management*, Springer-Verlag, New York, 1986.

7. Rossmann, Alain, "The Macintosh User Interface," *Outside Macintosh*, Addison-Wesley Publishing Co., New York, 1985.

8. Schmucker, K.J., *Object-Oriented Programming for the Macintosh*, Hayden Books, New York, 1986.

9. Apple Computer, Inc., *Inside Macintosh*, Addison-Wesley Publishing Co., New York, 1984.

10. Contingency Planning Subtask, Joint Theater-Level Simulation, *Software Engineering Maintenance Manual, Volume V, Model Interface Program(MIP)*, Jet Propulsion Laboratory, Pasadena, California, 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	CDR Joseph S.Stewart II, Code 55 Naval Postgraduate School Monterey, California 93943-5000	2
4.	Maj Thomas J. Brown, Code 39 Naval Postgraduate School Monterey, California 93943-5000	2
5.	C3 Academic Group, Code 39 Prof Michael G. Sovereign Naval Postgraduate School Monterey, California 93943-5000	2
6.	OJCS/J8 LtCol Richard H. Duff The Pentagon Washington, D.C. 20301	2
7.	Capt Stephen L. Lower 2202 Elm Street St. Joseph, Missouri 64505	2
8.	Rolands & Associates Corp. 500 Sloat Avenue Monterey, California 93940	1
9.	CDR Gary Porter Tactical Training Group Atlantic FLTCLANT Gallery Hall Dam Neck Virginia Beach, Virginia 23461	1

ATE
MED
8